

A First Course in Artificial Intelligence

About the Author



Deepak Khemani is Professor at Department of Computer Science and Engineering, IIT Madras.

He completed his B.Tech. (1980) in Mechanical Engineering, and M.Tech. (1983) and PhD. (1989) in Computer Science from IIT Bombay, and has been with IIT Madras since then. In between, he spent a year at Tata Research Development and Design Centre, Pune, and another at the youngest IIT at Mandi. He has had shorter stays at several computing departments in Europe.

Prof. Khemani's long-term goals are to build articulate problem solving systems using AI that can interact with human beings. His research interests include Memory Based Reasoning, Knowledge Representation and Reasoning, Planning and Constraint Satisfaction, Qualitative Reasoning and Natural Language Processing.

A First Course in Artificial Intelligence

Deepak Khemani

*Department of Computer Science and Engineering Indian Institute
of Technology (IIT) Madras Chennai*



McGraw Hill Education (India) Private Limited
NEW DELHI

McGraw Hill Education Offices

New Delhi New York St Louis San Francisco Auckland Bogotá
Caracas Kuala Lumpur Lisbon London Madrid Mexico City Milan
Montreal San Juan Santiago Singapore Sydney Tokyo Toronto



McGraw Hill Education (India) Private Limited

Published by McGraw Hill Education (India) Private Limited
P-24, Green Park Extension, New Delhi 110 016

Sales Territories: India, Pakistan, Nepal, Bangladesh, Bhutan and Sri Lanka

A First Course in Artificial Intelligence

Copyright © 2013, by Deepak Khemani

No part of this publication may be reproduced or distributed in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise or stored in a database or retrieval system without the prior written permission of the publishers. The program listings (if any) may be entered, stored and executed in a computer system, but they may not be reproduced for publication.

This edition can be exported in Indian Subcontinent by the publishers, McGraw Hill Education (India) Private Limited

Print Edition:

ISBN-13: 978-1-25-902998-1

ISBN-10: 1-25-902998-0

Ebook Edition:

ISBN-13: 978-93-832-8678-2

ISBN-10: 93-832-8678-4

Vice President and Managing Director: *Ajay Shukla*

Head—Higher Education (Publishing and Marketing): *Vibha Mahajan*

Publishing Manager (SEM & Tech. Ed.): *Shalini Jha*

Assistant Sponsoring Editor: *Smruti Snigdha*

Editorial Researcher: *Sourabh Maheshwari*

Manager—Production Systems: *Satinder S Baveja*

Assistant Manager—Editorial Services: *Sohini Mukherjee*

Senior Production Manager: *P L Pandita*

Assistant General Manager (Marketing)—Higher Education: *Vijay Sarathi*

Senior Product Specialist: *Tina Jajoriya*

Senior Graphic Designer—Cover: *Meenu Raghav*

General Manager—Production: *Rajender P Ghansela*

Manager—Production: *Reji Kumar*

Information contained in this work has been obtained by McGraw Hill Education (India), from sources believed to be reliable. However, neither McGraw Hill Education (India) nor its authors guarantee the accuracy or completeness of any information published herein, and neither McGraw Hill Education (India) nor its authors shall be responsible for any errors, omissions, or damages arising out of use of this information. This work is published with the understanding that McGraw Hill Education (India) and its authors are supplying information but are not attempting to render engineering or other professional services. If such services are required, the assistance of an appropriate professional should be sought.

Typeset at The Composers, 260, C.A. Apt., Paschim Vihar, New Delhi 110 063 and printed at

Cover Printer:

*For,
the interested student*

Table of Contents

Preface

Acknowledgements

Chapter 1 Introduction

- 1.1 Artificial Intelligence
- 1.2 Historical Backdrop
- 1.3 What is Intelligence?
- 1.4 The Bottom Line
- 1.5 Topics in AI

Points to Ponder

Chapter 2 State Space Search

- 2.1 Generate and Test
- 2.2 Simple Search 1
- 2.3 Depth First Search (DFS)
- 2.4 Breadth First Search (BFS)
- 2.5 Comparison of BFS and DFS
- 2.6 Quality of Solution
- 2.7 Depth Bounded *DFS* (*DBDFS*)
- 2.8 Depth First Iterative Deepening (*DFID*)

Exercises

Chapter 3 Heuristic Search

- 3.1 Heuristic Functions
- 3.2 Best First Search
- 3.3 Hill Climbing
- 3.4 Local Maxima
- 3.5 Solution Space Search
- 3.6 Variable Neighbourhood Descent
- 3.7 Beam Search
- 3.8 Tabu Search
- 3.9 Peak to Peak Methods
- 3.10 Discussion

Exercises

Chapter 4 Randomized Search and Emergent Systems

Escaping Local Maxima

4.1 Iterated Hill Climbing

4.2 Simulated Annealing

4.3 Genetic Algorithms

4.4 The Travelling Salesman Problem

4.5 Neural Networks

4.6 Emergent Systems

4.7 Ant Colony Optimization

4.8 Discussion

Exercises

Chapter 5 Finding Optimal Paths

5.1 Brute Force

5.2 Branch & Bound

5.3 Refinement Search

5.4 Dijkstra's Algorithm

5.5 Algorithm A*

5.6 Admissibility of A*

5.7 Iterative Deepening A* (IDA*)

5.8 Recursive Best First Search (RBFS)

5.9 Pruning the *CLOSED* List

5.10 Pruning the *OPEN* List

5.11 Divide and Conquer Beam Stack Search

5.12 Discussion

Exercises

Chapter 6 Problem Decomposition

6.1 *SAINT*

6.2 *Dendral*

6.3 Goal Trees

6.4 Rule Based Systems

6.5 *XCON*

6.6 Rule Based Expert Systems

6.7 Discussion

Exercises

Chapter 7 Planning

- 7.1 The *STRIPS* Domain
- 7.2 Forward State Space Planning
- 7.3 Backwards State Space Planning
- 7.4 Goal Stack Planning
- 7.5 Plan Space Planning
- 7.6 A Unified Framework for Planning
- 7.7 Discussion

Exercises

Chapter 8 Game Playing

- 8.1 Board Games
- 8.2 Game Playing Algorithms
- 8.3 Limitations of Search
- 8.4 Other Games
- 8.5 Beyond Search
- 8.6 Discussion

Exercises

Chapter 9 Constraint Satisfaction Problems

- 9.1 *N*-Queens
- 9.2 Constraint Propagation
- 9.3 Scene Labelling
- 9.4 Higher Order Consistency
- 9.5 Directional Consistency
- 9.6 Algorithm Backtracking
- 9.7 Lookahead Strategies
- 9.8 Strategic Retreat
- 9.9 Discussion

Exercises

Chapter 10 Advanced Planning Methods

- 10.1 GraphPlan
- 10.2 Planning as Constraint Satisfaction
- 10.3 Planning as Satisfiability
- 10.4 Heuristic Search
- 10.5 Durative Actions

- 10.6 Trajectory Constraints and Preferences
- 10.7 Planning in the Real World
- 10.8 Discussion
- Exercises*

Chapter 11 Knowledge Based Reasoning

- 11.1 Agents
- 11.2 Facets of Knowledge

Chapter 12 Logic and Inferences

- 12.1 Formal Logic
- 12.2 History of Logic and Knowledge
- 12.3 Propositional Logic
- 12.4 Resolution Method in Propositional Logic
- 12.5 First Order Logic
- 12.6 Incompleteness of Forward Chaining
- 12.7 Resolution Refutation in *FOL*
- 12.8 Deductive Retrieval
- 12.9 Complexity of Resolution Method in *FOL*
- 12.10 *Horn* Clauses and *SLD* Resolution
- 12.11 Backward Chaining
- 12.12 Second Order Logic
- 12.13 Discussion
- Exercises*

Chapter 13 Concepts and Language

- 13.1 The Conceptual Domain: The Ontological Base
- 13.2 Reification
- 13.3 *RDF* and the Semantic Web
- 13.4 Properties
- 13.5 Event Calculus
- 13.6 Conceptual Dependency Theory
- 13.7 Conceptual Analysis
- 13.8 Discussion
- Exercises*

Chapter 14 Structured Knowledge Representations

- 14.1 Hierarchies in the Domain

- 14.2 The Schema
- 14.3 Frames
- 14.4 The Semantic Net
- 14.5 Scripts, Goals, Plans and MOPs
- 14.6 Inheritance in Taxonomies
- 14.7 Description Logics
- 14.8 Formal Concept Analysis
- 14.9 Conceptual Graphs
- 14.10 Discussion
- Exercises*

Chapter 15 Memory and Experience: Case Based Reasoning

- 15.1 Case Based Reasoning
- 15.2 Retrieval
- 15.3 Reuse and Adaptation
- 15.4 Discussion
- Exercises*

Chapter 16 Natural Language Processing

- 16.1 Classic Problems in NLP and Schools of Thought
- 16.2 Basic NLP Techniques
- 16.3 Applications
- 16.4 Natural Language Generation
- Exercises*

Chapter 17 Reasoning Under Uncertainty

- 17.1 Default Reasoning
- 17.2 Qualitative Reasoning
- 17.3 Model Based Diagnosis
- 17.4 Assumption Based Reasoning and Truth Maintenance
- 17.5 Probabilistic Reasoning
- 17.6 Stochastic Actions
- 17.7 Combining Evidences to form Beliefs
- 17.8 Discussion
- Exercises*

Chapter 18 Machine Learning

18.1 Naïve Bayes Classifiers

18.2 Inference in Bayesian Networks

18.3 Hidden Markov Models

18.4 Concept Learning

18.5 Decision Trees

18.6 The *K-means* Clustering Algorithm

18.7 Learning from Outcomes

18.8 Artificial Neural Networks

18.9 Discussion

Exercises

Epilogue

References

Index

Preface

Artificial Intelligence (AI) is a confluence of many disciplines. There are two distinct strands of AI activity. One, a more cognitive approach, seeks to understand how intelligent behaviour arises. For researchers pursuing this flavour, AI provides a computational platform to test their ideas. The other strand adopts an engineering approach, and the goal is to construct intelligent machines. What the two have in common is that the ideas manifest themselves in computer programs. And these programs draw upon ideas from many disciplines—computer science, philosophy, psychology, economics, mathematics, logic and operations research.

The problem solving viewpoint says that intelligence is the ability to solve problems. We often refer to an autonomous program that senses its environment and acts independently in a goal directed manner as an agent. Then, given a state in which a problem solver exists, and given a set of actions that the problem solving agent has access to, the intelligent agent chooses the actions that will result in the agent being in a desired state that satisfies the agent's goals. Within this broad framework, all kinds of problems can be posed, and many different kinds of techniques can be applied. Identifying a desired state, or goal of an agent, is also within the scope of intelligence, though often categorized as wisdom, exemplified by the phrase "wise choice".

There are two broad approaches to solving problems. The first is to treat every problem to be solved using first principles, and the second is to harness knowledge gleaned from experience or from other agents. The first principles approach says that an agent can solve a problem by reasoning about actions, exploring combinations, and choosing the ones that lead to the solution. Even this exploration can be done in an informed manner. We begin our own exploration of AI by first studying this approach. However, as we soon see, there lurks the danger of combinatorial explosion that the agent has to contend with. We then refine and modify the search based approach to include heuristic knowledge, and then we move towards ways to deploy more explicit forms of domain specific knowledge. These will include logic and reasoning, memory structures and the exploitation of experience, deeper knowledge in models and ontology, and the relation between language and knowledge. We will look at machine learning techniques that can learn from instances of data. Along the way, we also look at game playing and planning problems.

Our study of knowledge begins with employing logic as a vehicle. We draw upon the strong mathematical and philosophical base of logical reasoning. Here again, we witness that the weak methods for theorem proving encounter the feared adversary, CombEx, or combinatorial explosion. We look at ways to structure knowledge in an effort to connect

the related pieces and make compact representations. Our study of ontology and description logic is also driven by the need for programs to communicate and exchange information by different machines across the Internet. Humans use language not only to communicate but also to represent knowledge, for example in a book. Formal reasoning and argumentation is also often done using language. We devote three chapters to dealing with various aspects of language, including methods for text processing which have gained prominence with the explosion of information available online. I have been sometimes asked by students as to why I teach, and write about, approaches like the conceptual dependency theory and scripts etcetera, which were popular up till the early eighties in the last century and not really pursued after that. The reason is that semantics is important, as also is pragmatics that involves the representation and reasoning with stereotypical knowledge. If a computer is to interact *meaningfully* with a user then it *will* need to access the meaning of what is being said. The specific formalism that is used is not the important thing, but the *idea* that the meaning of natural language sentences is expressed in a language that is used to model the concepts in the underlying domain. With increasing computational power and advances in representation and reasoning, such approaches to handle deep knowledge are bound to make a comeback.

However, the use of classical mathematical logic is inadequate for reasoning about a changing world and where we have to contend with incomplete information. Mathematical logic connects true statements to other true statements. It is not quite equipped to talk about statements whose truth values are neither unequivocally true nor unequivocally false. Researchers have tried various approaches to handle uncertain knowledge, from probabilistic methods to characterizing conditions under which statements are true, to prescribing degrees of belief. We look at some of these in the latter half of the book.

As we move from pure search based methods to more and more use of knowledge, we study the algorithms that have been devised at each stage. Eventually, a truly agile problem solving agent will need to integrate many of these algorithms for solving problems into one system.

About the Book

This book is a first course in artificial intelligence. The word *course* here is in the sense of offering, implying that the reader could go on to more detailed offerings in selective areas. It is like the first course of a sumptuous feast, which indeed artificial intelligence promises to be. Every chapter in the book, and in some case even sections in chapters, has enough material to write a book on, and in most cases indeed have books written on them. Our goal in this book is to provide in one place an introduction to the *core* concepts of artificial intelligence. If these ideas have to be embedded in an agent, whether physical or virtual then other

disciplines will be involved for sensing and acting. A physical intelligent agent, for example, may need visual, auditory and tactile sensors and a robotic platform or a speech synthesis system for effecting actions. The related disciplines of speech processing, computer vision and robotics are beyond the scope of this book. We confine ourselves to the cognitive aspects of intelligent agents.

The book is meant primarily for the first-time student, most likely in an undergraduate or even postgraduate course. But it could also be useful for a seasoned professional making a foray into this exciting field. That could be a software engineer with a desire to implement smarter programs (apps?) or a researcher from another domain keen to exploit artificial intelligence techniques in her area of activity. No background is required except familiarity with programming. Even for those who do not program themselves, the book could provide insights into software written by others. And finally, I hope this book will be a part of essential reading for the budding artificial intelligence researcher.

The book has evolved over a period of over twenty years of teaching artificial intelligence at IIT Madras, and is based on a set of six elective courses taught by my colleagues and me in the department—Introduction to AI, Knowledge Representation and Reasoning (KRR), Planning and Constraint Satisfaction (PCS), Memory Based Reasoning in AI (MBR in AI), Natural Language Processing (NLP), and Machine Learning (ML). In fact, I am privileged to have my colleague Sutanu Chakraborti who teaches NLP write Chapter 16, and Ashish Tendulkar who taught ML contribute to Chapter 18. In addition, there is material on Qualitative Reasoning, Probabilistic Reasoning and Artificial Neural Networks. The last two also have courses devoted to them. Most of the book reflects the way I teach the AI related courses, and contains examples and figures designed to aid the reader in understanding concepts. The book has a narrative style, telling the story of the quest for AI, in which concepts may sometimes be expressed again in different words. I have also received feedback that this reinforces the learning of concepts.

The book follows a theme of building intelligent systems from scratch. We begin with general purpose search methods and gradually bring out the need for knowledge for problem solving, which first appears in the form of heuristic functions before appearing in the form of explicit symbolic structures designed to address the different issues in reasoning, eventually even in an uncertain world with incomplete information. On the way, we look at alternate approaches like constraint satisfaction and also at specific problems like planning and game playing. As we go along, various sub-areas of artificial intelligence emerge.

This book is an attempt to give a comprehensive account of Artificial Intelligence to the reader in one place. I would expect the serious reader to complement this material with other literature, much of which has been indicated in the text.

The book has a flow of chapters building up a case for and leading up

to future chapters. However, an attempt has been made to write each chapter so that it can be read in isolation, and make it possible for an instructor to choose and string together chapters to form a course.

Roadmap to the Syllabus

The book is not written explicitly for a one-semester course or a one-year course. In fact, it would be difficult to include all the material in a one- or even two-semester course, unless the syllabus is carefully selected from different chapters. We assume each teacher will formulate a course syllabus that fits in well in the curriculum, and encourage the interested student to read up the remaining part. Even the order in which the material is taught is not sacrosanct, though there are some dependencies. For example, it has been suggested by a reviewer or two that the chapter on games can come earlier in the book. In fact, this is the case in the introductory course I teach on AI, and it does lead to a programming assignment that can be given early—writing game playing programs that participate in a course tournament with marks being earned by the programs—that students find quite exciting.

The following contents could form a *one-semester course*.

Chapters 1–4. Chapter 5 up to 5.7, Chapter 7, Chapter 8 up to 8.2, Section 1 from Chapter 10, Chapters 11–12, Chapter 16 and some parts of Chapter 18.

A *three-semester course* would do more justice to the subject and could be as follows.

Semester 1. Problem solving using search. Chapters 1–4. Chapter 5 up to 5.7, Chapter 6–9, Section 1 from Chapter 10, Chapter 12.

Semester 2. Knowledge representation and reasoning. Chapter 11, review of Chapter 12, Chapter 6, Chapter 13 up to Section 5, Chapter 14 up to Section 7, Chapter 15.

Semester 3. Natural language processing, handling uncertainty, and machine learning. Chapters 16–18.

A *two-semester course* could cut down on some material from the second semester of the three-semester syllabus, and add some from the third.

Online Learning Center

The Online Learning Center is accessible at <http://www.mhhe.com/khemani/ai> and contains

- Wiki page for programs
- PowerPoint slides of the chapters
- Solution Manual with pointers
- Web links to additional reading

A Final Word

The proof of the pudding is in the eating. Learning artificial intelligence has to be reflected in the implementation of programs that embody the algorithms written in the book. The algorithms have deliberately not been written in a programming language. One reason for this is that this allows us to abstract away from a programming language and focus on the algorithms. Another is that the algorithms are indeed independent of the language they can be implemented in. While someone might like to use a language like Lisp or Haskell, other readers might prefer C, C++, Python or Java. We leave the choice to the user. A third reason, as appreciated by a reviewer, is that this allows the instructor to specify the algorithm with just enough detail, and actually get the students to learn by doing the implementation. Many of these algorithms have been implemented by students at IIT Madras over the years as part of course assignments. Perhaps we can use some of those programs to seed a Wiki site for algorithms in which readers can contribute implementations in different flavours. Do look out on the book webpage for a link.

Feedback from the readers is welcome at AFCAI@cse.iitm.ac.in

Deepak Khemani

Publisher's Note

Do you have any further request or a suggestion? We are always open to new ideas (the best ones come from you!). You may send your comments to tmh.csefeedback@gmail.com

Acknowledgements

This book is a retelling in my words the story of artificial intelligence with which I have been involved with as a researcher and as a teacher at IIT Madras. First and foremost, I am indebted to all the authors of many books and research papers whose work this book draws upon. And what has been written owes itself to the society of thinkers who have imagined and conjured up in symbolic terms the world of the mind: Thinking about thinking.

There are far too many to whom I am indebted for the appreciation and passion I have developed for the field of artificial intelligence. In the following lines, I only name a few who have in some direct way contributed to the book.

Every batch I have taught here has had some brilliant students who have made teaching the subject a joy. I am grateful that they were in my class. The ever-helpful librarian in our department, R Balasundaram, helped in many ways including procuring books speedily.

A host of people have provided reviews and feedback on the manuscript at various stages. I am grateful to the following, in no particular order, for reviewing parts of the manuscript and providing invaluable feedback that helped weed out many errors and inconsistencies in the book—Dasarathi S, Delip Rao, Rajat Bhattacharjee, Pushpak Bhattacharya, M Narasimha Murty, Vipin B S (who also helped bring uniformity to the algorithms), Baskaran Sankaranarayanan, Dipti Deodhare, I Murugeswari, Sutanu Chakraborti, Jyothis V, Anand Kumar, Yaji Sripada, Bharat Ranjan Kavuleri, B Radhika Selvamani, Sonal Oswal, Nitin Dhiman, D R Lakshminarasimhaiah, Dileep A D, Sarath Chandar, Alexandra Weber, Abhishek Ghose, Sanchit Khattri, Anil Khemani, N S Narayanaswamy, Geeta Raman, and Deepak S Padmanabhan. I apologize to those whose names I have missed. I would also like to thank the named and unnamed reviewers who looked at earlier versions of the manuscript including, but not limited to, Madhavan Mukund, Denise Penrose, Sharvani G S, Christian Jacob, Joseph Lewis, Art Shindhelm, Son Cao Tran, Pabitra Mitra, A Kannan, and Ian Watson.

Thanks are also due to the following reviewers who reviewed the manuscript at the behest of the publisher - Kumkum Garg (IIT Roorkee), Krishna Asawa (Jaypee Institute of Information Technology, Noida), Sameer Bhawe (MHOW, Indore), P J Kulkarni (Walchand College of Engineering, Sangli), Deipali V Gore (People's Education Society (PES) Modern College of Engineering, Pune), L M R J Lobo (Walchand Institute of Technology, Solapur), Nilesh J Uke (Sinhgad College of Engineering, Pune), S N Sivanandam (PSG College of Technology, Coimbatore), E Grace Mary Kanaga (Karunya University, Coimbatore), S G Sanjeevi

(NIT Warangal), Y R Ramesh (NIT Warangal), Chittranjan Hota (BITS Pilani—Hyderabad Campus), S Kavitha (SSN College of Engineering, Kalavakkam), Cheng-Seen Ho (National Taiwan University of Science and Technology, Taipei), Von-Wun Soo (National Tsing Hua University, HsinChu).

All the reviewers mentioned above have only seen the book in small parts. It has all come together now thanks to the publishing team. I am grateful to Smruti Snigdha and her team at McGraw Hill Education (India) who provided all kinds of help and listened patiently to my opinions on the production aspects. The book you are holding owes much to them.

I am grateful to the IIT Madras Golden Jubilee Book Writing Scheme, which allowed me a semester of sabbatical from teaching in 2007 to write this book. I had been writing for years and thought I was close to finishing it then. I did, as you can see, eventually complete it (complying in fact with Hofstadter's Law¹). The beautiful IIT Madras campus with its banyan trees, deer and monkeys, has been a wonderful place to live and work in. I should not fail to mention the enjoyable one year I spent at IIT Mandi in the sylvan and scenic Beas valley, amongst the lovely people of Himachal, without which the book would have probably come out one year earlier.

I am grateful to many friends and family who have contributed in many ways to my book. In particular, I should mention R Krishna Kumar, K S Balaji and Prakash Paranjape for urging me at all times to get on with it.

Finally, I owe everything to my mother, my past, and my daughter, my future.

Deepak Khemani

¹ Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law. — Douglas Hofstadter, Gödel, Escher, Bach: An Eternal Golden Braid (1979).

Introduction

Chapter 1

The human mind is a wondrous thing. For centuries, humans have reflected upon the world we live in; looking at the sun, the moon and the stars in wonderment, observing nature as the days grew and shrunk, seasons change, the searing heat of the summer followed by the dark clouds bringing welcome rain; observing life around us in the form of plants and trees and rabbits and bees; the constant struggle for food and the desire to avoid becoming food; and most of all, upon themselves and the human species amidst all these.

We, the *Homo sapiens*, have learned to exploit this ability to reflect upon the world around us to make life better for ourselves. After the extinction of the dinosaurs, when mammals began flourishing, we were probably just one amongst the many species foraging for food and fleeing the predators. Evolution, however, bestowed us with bigger brains; and we started changing the rules of the survival game being played amongst the different species. We learnt to form societies and evolved language as a means of communication between individuals; we learnt to protect and domesticate species of flora and fauna that were advantageous for us; and perhaps most importantly, we learnt to make and use tools.

As humankind became technologically more advanced, individual lifestyles diversified enormously. Though we are all born the same, as we grow and learn we channelize our activity and occupations into a multitude of directions, and the more technologically advanced we have become, the more specialized our professions have become. This has no doubt given us the benefits of efficiency, exemplified by workers in a factory, each doing a specific task. We drink our coffee grown by someone, transported by someone else; roasted, ground, and packaged by an organized mini-society, and sold to us by someone else. Everything we use in our daily lives goes through a similar process, and we are able to enjoy a rich and bountiful existence¹ because each of us is occupied with doing something well, for the benefit of many others.

Technological advancement combined with an organized society has created spaces in which sections of society can devote their energies towards things beyond our basic necessities. Humankind has developed music, literature, painting, sculpture, theatre and many other art forms to cater to our minds. Research in science and society pushes the frontier of our understanding further and further. Technological development has accelerated with this deeper understanding.

And during all stages of human development, our minds have continued to reflect on life around us and upon ourselves. We have continuously asked questions about our own existence and about our own minds. On this quest for understanding our minds, philosophers have now been joined by mathematicians, and psychologists, and economists, and cognitive scientists, and most recently, computer scientists. We not only want to understand minds, brains and intelligence, but also want to create minds, brains and intelligence. This quest goes broadly under the name of artificial intelligence.

1.1 Artificial Intelligence

The field of artificial intelligence (AI) that we explore in this book came into being as soon as digital computers were built. But even before the electronic machine came into existence, Gottfried Leibniz (1646–1716) and Blaise Pascal (1623–1662) had explored the construction of mechanical calculating machines. Charles Babbage (1791–1871) had designed the first stored program machine. The ability of storing and manipulating symbols evoked the possibility of doing so intelligently and autonomously by the machine; and artificial intelligence became a lodestar for the pioneers of computing.

The name *artificial intelligence* is credited to John McCarthy who, along with Marvin Minsky and Claude Shannon (1916–2001), organized the *Dartmouth Conference* in 1956. The conference was to be a “two month, ten-man study of artificial intelligence ... on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described, that a machine can be made to simulate it.”

The above goal set the tone for the definition of artificial intelligence in the textbooks that appeared on the subject (see for example (Feigenbaum and Feldman, 1963), (Nilsson, 1971), (Newell and Simon, 1972), (Raphael, 1976), (Winston, 1977), (Rich, 1983) and (Charniak and McDermott, 1985)). Some definitions focus on human intelligence, and some on hard problems.

- We call programs ‘intelligent’, if they exhibit behaviours that would be regarded intelligent if they were exhibited by human beings—Herbert Simon.
- Physicists ask what kind of place this universe is and seek to characterize its behaviour systematically. Biologists ask what it means for a physical system to be living. We (in AI) wonder what kind of information-processing system can ask such questions—Avron Barr and Edward Feigenbaum (1981).
- AI is the study of techniques for solving exponentially hard problems in polynomial time by exploiting knowledge about the problem domain—Elaine Rich.
- AI is the study of mental faculties through the use of computational

models—Eugene Charniak and Drew McDermott.

Even while Charles Babbage was designing a mechanical computer, his collaborator Lady Ada Lovelace (1815–1852) had prudently observed that a machine can only do what it is programmed to do. This perhaps was an indicator of the debates that accompanied the foray into artificial intelligence (discussed briefly in a section below) which led many people to assert that the goal of AI was to mimic human intelligence. We should then perhaps add our preferred description of what AI is all about:

- The fundamental goal of this research is not merely to mimic intelligence or produce some clever fake. “AI” wants the genuine article; *machines with minds*—John Haugeland (1985).

Haugeland also says that he would have preferred the name *Synthetic Intelligence*, since for some people, the word *artificial intelligence* has a connotation of not being real. Other names that have been suggested are *Applied Epistemology*, *Heuristic Programming*, *Machine Intelligence* and *Computational Intelligence*. But it is *Artificial Intelligence* that has stuck both within the scientific community and in popular imagination.

The field of AI is a culmination of a long series of efforts to build sophisticated machinery in Europe over the last few centuries, along with advances in philosophy, mathematics and logic in the scientific community. Perhaps a quick look at the history of the activity leading up to our modern times will be insightful. The serious student of artificial intelligence is referred to the books *Machines Who Think* by Pamela McCorduck (1973), and *AI: The Very Idea* by John Haugeland (1985) that explore the historical and philosophical background in considerably more detail. The material in the next section draws significantly from them.

1.2 Historical Backdrop

The urge to create intelligent beings can be traced back to Greek mythology. Hephaestus, son of Hera (the queen of gods and Zeus' wife), constructed humanlike creations regularly in his forge. As a present from Zeus to Europa (a human princess), he created Talos out of bronze to guard and defend the island of Crete, where Europa lived. Perhaps his most famous creation is *Pandora*. He did so at the behest of Zeus who wanted to punish humankind for accepting Prometheus's gift of fire. Pandora is sent to Earth with a casket, which she has been forbidden to open, but does so, overcome by curiosity, and releases the world's evils.

Disenchanted with human women, Pygmalion is said to have created Galatea out of ivory, and fell in love with his own creation. Aphrodite, the goddess of love, obliged him by breathing life into this man made woman. Such stories in which creations of men that come alive abound in literature.

One of the earliest mechanical contraptions actually built was by Heron of Alexandria in the first century ad when he built water powered mechanical ducks that emitted realistic chirping sounds.

Gradually, as metal-working skills improved, the line between fact and fiction got blurred. The European people were not as affluent as they are now, and many were totally occupied with eking out a living and protecting themselves from the harsh winters. Science, and art, was the forte of the rich aristocrats, and scientists had begun to gain liberty of action and were beginning independent philosophical and scientific investigations. The works of scientists and artisans (and sorcerers) blended with folklore and often became a subject matter of folklore itself.

Daedalus, most famous for his artificial wings, is also credited with creating artificial people. In mediaeval Europe, Pope Sylvester II (946–1003) is said to have made a statue with a talking head, with a limited vocabulary, and a penchant for predicting the future. It gave replies to queries with a *yes* or a *no*, and its human audience did not doubt that the answer was preceded by some impressive mental activity. Arab astrologers were believed to have constructed a thinking machine called the *zairja*. The *zairja* caught afire imagination of a missionary, Ramon Lull (1232–1315), with religious zeal, who decided to build a Christian version called the *Ars Magna*. Like the *zairja*, the *Ars Magna* was constructed using a set of rotating discs and was aimed “to bring reason to bear on all subjects and, in this way, arrive at the truth without the trouble of thinking or fact finding” (McCorduck, 1973).

This period also marks the emergence of the art of making elaborate clocks decorated with animated figures, which helped establish the belief that learned men kept artificial servants. In the sixteenth century, the rabbi Judah ben Loew (1520–1609) is said to have created an artificial man named Joseph Golem, who could only be instructed properly by the rabbi. Earlier in the century, Paracelsus (1493–1541), a physician by profession, is reputed to have created a *homunculus*, or a little man.

The art of mechanical creatures flourished in the following years. The Archbishop of Salsburg built a working model of a complete miniature town, operated by water power. In 1644, the French engineer, Isaac de Caus, designed a menacing metal owl along with a set of smaller birds chirping around it, except when the owl was looking at them. In the eighteenth century, Jacques de Vaucanson (1709–1782) of Paris constructed a mechanical duck that could bend its neck, move its wings and feet, and eat and apparently digest food. Vaucanson himself was careful not to make strong claims about the duck’s innards, asserting that he was only interested in imitating the larger aspects. This approach has been echoed often in artificial intelligence. For example, chess playing machines do not make decisions the way human players do. A creation of doubtful veracity was the chess automaton, the Turk, created by the Hungarian baron Wolfgang von Kempelen (1734–1804). The contraption was demonstrated in many courts in Europe with great success. The automaton played good chess, but that was because there was a

diminutive man inside the contraption.

Fiction took the upper hand once again and Mary Shelley (1797–1851), an acquaintance of Lord Byron (1788–1824), wrote the classic horror novel *Frankenstein*, in which the character Dr Frankenstein, creates a humanoid which turns into a monster. The word *robot*, which means *worker* in Czech, first appeared in a play called *Rossum's Universal Robots* (RUR) by Karel Čapek (1921) (1890–1938); apparently on the suggestion of his brother Josef (Levy, 2008). It is derived from the word *robota* meaning *forced labour*, and the robots in Čapek's play were supposed to be machines resembling humans in appearance and meant to do their master's bidding. The 'Frankenstein' theme was repeated in Čapek's play in which the robots rebel and destroy the human race. The prospect of human creations running haywire has long haunted many people. Isaac Asimov who took up the baton of writing about (fictional) artificial creatures, formulated the three laws of robotics in which he said the following:

A robot cannot injure a human being, or through inaction allow a human being to come to harm. A robot must obey any orders given to it by human beings, except where such orders would conflict with the First Law.

A robot must protect its own existence as long as such protection does not conflict with the First or Second Law. These were laws in the civic sense that Asimov said should be hardwired into the "brains" of robots. A *zeroeth* law was subsequently added: A robot may not injure humanity, or, by inaction, allow humanity to come to harm.

But by then, Charles Babbage (1791–1871) had arrived. The son of a wealthy banker, he was exposed to several exhibitions of machinery, and it was his desire to build the ultimate calculating machine. In one such exhibition in Hanover, the young Babbage was much impressed by a man who called himself Merlin, in whose workshop he saw fascinating uncovered female figures in silver, one of which "*would walk about, use an eyeglass occasionally, and bow frequently*", and the other, Babbage wrote, "*was a dancer, full of imagination and irresistible*." In 1822, he constructed his smaller machine—the *Difference Engine*. His larger project, the *Analytic Engine*, however never found the funding it deserved. A description of the design can be found in (Menabrea, 1842)². The distinctive feature of the Analytic Engine was that it introduced the stored-program concept, for which we acknowledge Charles Babbage as the father of computing. The first realization in physical form of a stored-program computer was in fact the EDSAC built in Cambridge a century later. The sketch by Menabrea was translated by Lady Ada Augusta (1815–1852), Countess of Lovelace, daughter of Lord Byron, who worked closely with Babbage and is often referred to as the first programmer. The programming language *Ada* has been named after her. She has famously said that "*The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform*." She did recognize the possibility of representing other kinds of

things in the “calculating machine”, including letters of the alphabet, notes in music or even pieces on a chessboard.

1.2.1 Mind and Body

The notion of the *mind* came into our thoughts much later than minds came into existence. Almost all life forms sense the world, and most have eyes, arguably our principal means of perceiving the world around us. The earliest notions of the world were probably that the world was as we saw it. What you see is what there is. Somewhat like the WYSIWYG (what you see is what you get) text editors. It is not surprising that the earliest models of the world were geo-centric ones based on the perception of a flat earth (Figure 1.1).

This view of the world was drawn from the ideas of Greek philosophers Plato and Aristotle who dwelt at length upon the notion of ideas. For Plato, both an object in the world and the idea of that object were derived from a world of perfect ideas or ideals. Our own ideas, as well as the objects in the world out there, could be imperfect. Aristotle did away with the notion of ideals, and said that ideas were like pictures of the objects they were based on.

The ideas were true if they had the same form as the object. The human mind was thus something like a mirror reflecting the world to us. This, of course, takes “us” for granted. The more modern view is that the notion of “I” is itself a creation of our mind.

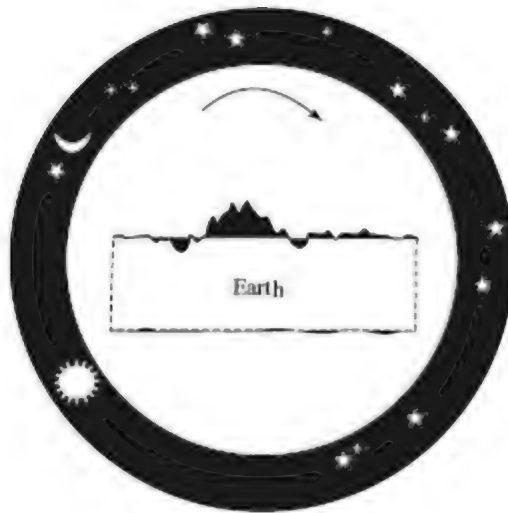


FIGURE 1.1 The world we saw was made up of a flat earth, with the heavens rotating around it.

It was, however, a challenge to make models of the heavens that could explain the apparently erratic motion of some heavenly bodies, the planets. It was Nicolaus Copernicus (1473–1543) who gave a simpler

model which provided a satisfactory explanation for the wanderings of the planets. According to his model, the daily movement of the sun, the moon and the stars was an illusion created by the rotation of the Earth, and that the Earth itself revolved around the sun. This view would have been an anathema to the prevailing human centric view of the world, and Copernicus delayed the publication of his book *On the Revolution of the Spheres* till the end of his life. When it did get published, the book drove a wedge between thought and reality. What we thought the world was, did not necessarily reflect as it really was.

Galileo Galilei (1564–1642) further developed the notion that what we think and perceive is something that happens within us. He said that the fact that when we taste, smell or hear something and associate a name with it, the name is really a name we give to the sensation that arises in us. In our modern parlance, one would say that the fragrance of a flower we perceive is caused by certain molecules emitted by the flower, but the fragrance we feel is the sensations caused by those molecules in our nasal glands. This notion was taken up later by John Locke (1632–1704) who distinguished between the *primary* qualities matter has, independent of the perceiver; and the *secondary* qualities of matter that are as perceived by us.

Galileo was perhaps one of the first to assert that we can reason with our own representations, which may not necessarily mirror what we are reasoning about, but stand for elements of the subject in some way. The most well known example of this is the way Galileo explained motion. Algebraic systems were yet to be devised, and Galileo's representations were geometric. If, for example, one side (the height) of a right angled triangle represents the final speed of a body accelerating uniformly from rest, and the other side (the base) represents the time elapsed then the area represents the distance covered. For uniform motion, the shape would be a rectangle. The important point is that Galileo showed that we can create representations, and that the representations can stand for something else. In fact, the same representation can stand for different things. We shall revisit this notion when we look at the semantics of First Order Logic later in the book.

It was the English philosopher Thomas Hobbes³ (1588–1679) who first put forward the view that *thinking is the manipulation of symbols* (Haugeland, 1985). Hobbes felt that thought was expressed in “phantasms” or thought “parcels”, and was clear and rational when it followed methodical rules. Galileo had said that all reality is mathematical, in the sense that everything is made up of particles, and our sensing of smell or taste was how we reacted to those particles. Hobbes extended this notion to say that thought too was made up of (expressed in) particles which the thinker manipulated. Hobbes had used the analogy of *thought parcels* with words in a language. However, he had no answer to the question of how a symbol can *mean* anything, because he had given up on the idea of thoughts being in the image of reality. That is a question that we can say is still unresolved. To some

extent now, we can say though that meaning arises via explicit association with, or an interpretation of, something that is known independently.

Hobbes *parcels* were material in nature. He would have been at home with the modern view of materialism in which one would think of the mind as something that emerges out of activity in the brain, which was the flow of electrons and the movement of neurotransmitters across synapses. It was René Descartes (1596–1650) who said that what the mind did was to manipulate *mental* symbols. Descartes was a pioneer in the development of algebraic systems. His analytic geometry showed an alternate symbolic way of representing lines and figures and reasoning with them. Galileo had shown that motion could be described using geometry. Descartes moved one more step away from the subject into the realm of mathematics. He said that (pure) mathematics is not concerned with any subject in particular, but with the kinds of relations that can arise in any subject of interest. This probably led to his notion of the mind as something that was concerned with manipulating symbols. He emphasized the fact that symbol systems were useful only when they came with methods and procedures for manipulating them and reasoning with them. Descartes was a *dualist*. He separated the mind from the body and said they were made of different kinds of “substances”. Minds had thoughts in them, while the physical universe operated by the physical laws of nature. This, however, led to the well known *mind-body* problem—if minds and bodies are distinct then how do they interact? When I think of typing this word, how does my thought influence my hand and fingers, which should have been behaving according to the laws of the physical world? Likewise, when the (light) photons from the garden impinge upon my retina, how do they trigger a corresponding thought about a flower?

The alternative to believing that minds are different from bodies is to believe that there is only one kind of thing. *Idealism*, discussed briefly later, says that (as far as we the thinkers can know) there are only ideas, and matter is something we imagine. The opposite of idealism is *materialism*, which says that only the material world exists and the mental world is a kind of a construct that (somehow) arises out of physical activity. This view would acknowledge our brains and what happens in them, but would have less to say about our minds, and even lesser about souls. Adapting this view, we still cannot escape the question of how could one *reason* or manipulate symbols *meaningfully*. Either we can manipulate them mechanically using some procedure, or we have to somehow bring meaning into the picture. If symbols are being manipulated meaningfully then who does the manipulation? One suggestion, mockingly put forward by its opponents, is that there is a *humunculus*, or a little man, who resides in our heads, reading our thoughts and creating new ones for us. Then of course, one would have to explain in turn how the humunculus works.

A way out was suggested by David Hume (1711–1776), an admirer of Isaac Newton (1644–1727). Hume put forth the idea that just as the

physical world operated according to the laws of nature, the mental world also operated according to its own laws, and did not need a manipulator. And just as we do not question why physical bodies obey the law of gravitation, we do not have to question why mental symbols will obey the laws that determine their manipulation. This does do away with the need for a manipulator, but Hume too could not answer the question of how the thought process could mean anything. Because meaning, as we now believe, can only be with respect to something, a real world, a domain in which the thinker operates.

While the debate over what minds are and whether machines can have minds of their own raged on, the arrival of the digital computer heralded an age when everyone plunged into writing programs to do interesting things without bothering about whether the programs would be called 'intelligent' or not.

1.2.2 AI in the Last Century

We take up the action in 1950 when Alan Turing proposed the *Imitation Game* described in the next section, and Claude Shannon published a paper on chess playing programs. Soon afterwards, checkers playing programs appeared, the first one being written in 1951 in the University of Manchester by Christopher Strachey (1916–1975). The program that became well known was by Arthur Samuel (1901–1990) which he demonstrated at the Dartmouth Conference in 1956. The basic components of a program to play checkers, or similar board games, are the search algorithm that peers into the future examining all variations, and the evaluation function that judges how good a board position is. Samuel's program is remembered as the first program that incorporated learning; it would improve with every game it played. This was achieved by tuning the parameters of the evaluation function based on the outcome of the game. This is important because the program chooses between alternative moves eventually based on the values returned by the evaluation function. The better the evaluation, the better would be the choice of the move. The program also remembered the values of board positions seen earlier. Samuel is also credited as being one of the people who invented *Alpha-Beta* pruning, a method that can drastically cut down on the search effort. In the end, the program became good enough to beat its own creator. One can imagine the kinds of fears that might have been raised amongst people after *Frankenstein* and *RUR*.

Computers then, by today's standard, were small (in memory size; physically they were huge) and slow. Computing was still an esoteric field.

The earliest proponents of artificial intelligence experimented with logic, language and games. Chess and checkers were already a fascination, and considered hallmarks of intelligence. Alex Bernstein had spoken about chess at Dartmouth, and developed the first chess playing

program in 1957. There was optimism amongst many that machines would soon be the best chess players around, but skepticism amongst others. Herbert Simon (1916–2001) and Alan Newell (1927–1992) said that a computer would be a world champion in ten years. In 1968, David Levy, an international master, wagered a bet that no machine would beat him in the next ten years, and won it by beating the strongest player at that time, from Northwestern University named *Chess4.7*. However, Levy later lost to *Deep Thought*⁴, originating from the Carnegie Mellon University, and in 1997, its successor *Deep Blue* from IBM beat the then reigning world champion, Gary Kasparov.

Game playing programs worked with a numerical evaluation function, and processing natural language beckoned as a new avenue of exploration. Noam Chomsky had been developing his theory of generative linguistics and moving on to transformational grammar. Armed with a grammatical model of language, and aided by a bilingual lexicon, machine translation became an area of research, much to the interest of the public. The earliest programs, however, were unable to contend with the richness of language, in particular when dealing with idioms. It was reported in the press that on translating “the spirit is willing but the flesh is weak” into Russian and back into English, the output read “the vodka is fine, but the meat has got spoiled” or something to that effect.

More interestingly, a natural language parser developed in 1963 by Susumu Kuno at Harvard revealed the degree of ambiguity in English language that often escapes human listeners, who latch on to one parse tree and the corresponding meaning. Given the input “*Time flies like an arrow*”, the parser produced at least four interpretations other than the one we normally assume. In 1966, the US government appointed *Automatic Language Processing Advisory Committee* (ALPAC) produced a negative report leading to a decline in research funding for machine translation. Like other ideas, it revived only with availability of increased computing power in the latter half of the century.

The task that computers were successful at from the word *go*, was logical reasoning. In the Dartmouth Conference in 1956, two (then) relatively unknown scientists from the West Coast of the US, Herbert Simon and Alan Newell, demonstrated a working theorem prover called LT (Logic Theorist), along with J C Shaw. The *Logic Theorist* proved many of the theorems in Russell and Whitehead’s *Principia Mathematica*, even finding shorter and elegant proofs for some of them. An attempt to publish a new proof in the Journal of Symbolic Logic however failed, apparently because a paper coauthored by a program was not acceptable! Another system that showed promise was a geometry theorem prover built by Gelernter in 1959. However, these theorem provers were based on search, and were faced with its nemesis—an exponentially growing search space. Like many AI problems, ‘geometry theorem proving’ too faced a revival many years later. In 1961, James Slage wrote the first symbolic integration program, *SAINT*, which formed the base for many symbolic mathematics tools.

Perhaps the most significant contribution of Newell and Simon was their program called GPS (General Problem Solver) that addressed general purpose problem solving, based on human thought processes. Their strategy called Means End Analysis (MEA) embodied a goal directed search strategy in which the problem solver repeatedly looks for methods (means) to achieve the most significant partial goal (ends), till all goals are solved. Their work found a home in the Carnegie Mellon University (CMU). It was first implanted in a production system language, OPS5, that was used to build expert systems. Subsequently, John Laird and Paul Rosenbloom built a general symbolic problem solving architecture known as SOAR, which is a popular tool now.

Neural networks, the emergent systems approach to problem solving that believes that the sum of interconnected simple processing elements is more than its part, too made their first appearance in the early years. Unlike classical AI systems that are designed and implemented in a top-down manner, neural networks are built by connecting the “neurons” according to a certain architecture, and then learning the weights of these connections by a process of training. It is in these weights, that all the knowledge gets captured, and it is generally not straightforward to interpret the weights in a meaningful manner. That is why we often call them *subsymbolic* systems. The first of these was a system called *Perceptron* built in 1957 by Frank Rosenblatt (1928–1971) at Cornell University. The Perceptron was essentially a single layer feed-forward neural network, and could learn to classify certain patterns. However, Minsky and Papert (1969) gave a mathematical proof that the Perceptron could handle only a simpler class of patterns, which proved to be a dampener for neural networks. It was only in the mid-eighties, with the revival of the Backpropagation algorithm for training multilayer neural networks by Rumelhart and Co. that research in neural network came again to the fore, and with bigger and faster machines available, was quite a rage amongst researchers in the nineties.

Meanwhile, John McCarthy, focused on logic in computer science, and proposed a system called *Advice Taker* in 1958, which was to use logic as a vehicle for knowledge representation and commonsense reasoning. He also invented *Lisp*, the programming language of choice for AI practitioners, based on Alonzo Church's *lambda calculus*. An important feature of Lisp was that a program could build data structures of arbitrary size dynamically during execution, which one believed was a great asset for AI programming. An interesting property of Lisp is that programs and data have the same representations, which means that programs can treat other programs, or even themselves, as data and modify them. Soon there was talk of building (dedicated) Lisp machines, mostly out of Massachusetts Institute of Technology, and a small number were even built by companies like Symbolics, Lisp Machines Inc., Xerox and Texas Instruments. However, an “AI winter” had set in with the drying of funds in the late eighties, largely due to the large hype that AI projects had generated only to disappoint. Meanwhile, personal computers were

growing by leaps and bounds, rendering Lisp machines, and even the language Lisp itself, to be a *forte* of a dedicated but small community.

In 1965, Alan Robinson published the *Resolution Method* for theorem proving that brought all kinds of logical inferences into one uniform fold. This approach gave a tremendous fillip to logical theorem-proving research. The notion of declarative knowledge representation took hold, and the possibility of using logic as a vehicle for writing programs emerged. As opposed to procedural languages that specify the execution steps, logic programming requires the programmer to specify the relations between the different entities involved in the problem. Collaborating with Robert Kowalski in Edinburgh University, Alain Colmerauer at the University II of Aix-Marseille at Luminy created the language *Prolog* in 1972. The idea of logic programming took off, mostly in Europe, and variants like Constraint Logic Programming, Parallel Logic Programming (with *Parlog*), Inductive Logic Programming and (much later) Linear Logic Programming were explored. The ideas were taken up with enthusiasm by the Japanese government, who announced the Fifth Generation Computer Systems (FGCS) project in 1982. The goal was to exploit massive amounts of parallelism and make logical inference the core of computing, measuring performance in LIPS (logical inferences per second), instead of the traditional MIPS (million instructions per second). Along with the Connection Machine—also designed to exploit massive parallelism, being built by Daniel Hillis around the same time, and almost in sync with the decline of the Lisp machine—the FGCS also faded away after the eighties.

The first autonomous robot was built at the Johns Hopkins University in the early sixties. The *Hopkin's Beast*, as it came to be known, was a simple minded creature, if it had a mind at all. The requirement of being standalone meant that it could not have a built in computer, which in those days would have been the size of a small room. It was battery driven, and its main goal was to keep the batteries charged. It ambled along the corridors of the department, occasionally stopping to plug itself into an electric outlet to “feed” itself. The first robot that could deliberate and do other stuff was built at the Stanford Research Institute over six years starting in 1966. It was named *Shakey*, apparently owing to “his” unsteady posture. A large computer nearby (there were no small ones then) analysed the data sensed by Shakey's TV camera, range finder and bump sensors, accepted commands in natural language typed into its console, and generated and monitored its plans.

In 2004, Shakey was inducted into CMU's Robotic Hall of Fame⁵ along with Honda's *ASIMO*; *Astroboy*, the Japanese animation of a robot with a soul; *C3PO*, a character from the “Star Wars” series; and *Robby the Robot* from MGM's *Forbidden Planet*. In the preceding year, the Mars Pathfinder Sojourner Rover, Unimate, R2-D2, and HAL 9000 were the first set of robots to be inducted. Sony's robotic dog, AIBO, joined them in 2006, along with the industrial robotic arm, SCARA. The entries from science fiction were *Maria* of *Metropolis* (1927), *David* of *Artificial*

Intelligence: AI (2001), and Gort of *The Day The Earth Stood Still* (1951). Lego's robotic kit *MINDSTORMS*, the agile *Hopper*, and the autonomous vehicle *NavLab 5* joined the elite group in 2008, along with *Lt. Cmdr. Data* from *Star Trek: The Next Generation*. Research in robotics is thriving, commercial development has become feasible, and we will soon expect robotic swarms to carry out search and rescue operations; and as we have observed elsewhere in this chapter, the possibility of robotic companions is no longer considered outlandish.

The first planning system *STRIPS* (Stanford Research Institute Planning System) was developed around the same time, 1971, by Richard Fikes and Nils Nilsson. So far, the work in planning had adopted a theorem proving approach. Planning is concerned with actions and change, and an important problem was to keep track of what is not changing. This was the well known *Frame Problem* described by John McCarthy and Patrick Hayes in 1969. The *STRIPS* program sidestepped the problem by doing away with time altogether in its representation, and only keeping the given state, making only those modifications that were the effects of actions. It was only later in the last decade of the twentieth century, that *time* made an appearance again in planning representations, as bigger and faster machines arrived, and other methods emerged. The *Frame Problem* is particularly difficult to deal with in an open-world model, where there may be other agencies. The *Circumscription* method given by McCarthy in 1980, laid the foundations of engaging in default reasoning in a changing world.

One system that created quite an impression in 1970 was Terry Winograd's *natural language understanding system*, *SHRDLU*⁶. It was written in the *Micro-Planner* language that was part of the *Planner* series of languages, that introduced an alternative to logic programming by adopting a procedural approach to knowledge representation, though they did have elements of logical reasoning. *SHRDLU* could carry out a conversation about a domain of blocks. It would listen to instructions like "*pick up the green cone*", and if this was ambiguous, it would respond with "*I don't understand which cone you mean*", and you could say something like "*the one on the red cube*", which it would understand. Then if you told it to find a bigger one and put it in the box, it would check back whether by "it" you meant the bigger one and not the one it was holding, and so on.

The optimism generated by *SHRDLU* generated expectations of computer systems that would soon talk to people, but it was not to be. Language is too rich a domain to be handled by simple means. Interestingly, Winograd's office mate and fellow research student at MIT, Eugene Charniak, had made the pertinent observation even then that the real task behind understanding language lies in *knowledge representation* and *reasoning*. This is the dominant theme when it comes to processing language that we explore in this book as well. Incidentally, Charniak's implementations were also done in *Micro-Planner*.

A similar view was expressed by Roger Schank and his students at

Yale University. They did a considerable amount of work, hand coding knowledge into complex systems that could read stories and answer questions about them in an “intelligent” fashion. This effort peaked in the eighties, with the program *BORIS* written by Michael Dyer. People whose research approach was to build large working systems with lots of knowledge were sometimes referred to as “scruffies”, as approached to the “neats” who focused on logic and formal algorithms. But the effort of carefully crafting complex knowledge structures was becoming too much, and by then, the pendulum was swinging in AI research towards “neat” general purpose, problem solving methods. Machines were getting faster; neural networks were on the ascendant; and approaches to knowledge discovery were showing promise. A decade or so later, even language research would start taking a statistical hue.

One of Schank’s ideas was that stories follow patterns, like *Scripts*, and that story understanding requires knowledge of such pattern structures. There were other people who investigated structured knowledge representations as well. A well known formalism is the theory of *Frames*, proposed by Marvin Minsky in 1974, which explored how knowledge exists in structured groups related to other frames by different kinds of relations. The idea of *Frames* eventually led to in the development of Object Oriented Programming Systems. A related idea was that of *Semantic Nets* proposed by Richard H Richens and developed by Ross Quillian in the early sixties. The approach of representing knowledge as a network of interconnected networks was taken up by John Sowa in his *Conceptual Graphs* in the eighties. And now the system *Wordnet*⁷ developed at Princeton University is a valuable resource for anyone interested in getting at the knowledge behind lexical utterances.

The importance of knowledge had been brilliantly illustrated by the program *Dendral*. Developed at the Stanford University by Edward Feigenbaum and his colleagues in 1967, *Dendral* was a program designed to be a chemist’s assistant. It took the molecular formula of a chemical compound, its mass spectrogram and searched through a vast number of possibilities to identify the structure of the molecule. It was able to do this effectively with the aid of large amounts of knowledge gleaned from expert chemists. This resulted in performance that matched that of an expert chemist. And the idea of Expert Systems was born: to educe domain specific knowledge and put it in the machine. The preferred form of representation was *rules*, and we often refer to such systems as *Rule Based Expert Systems*. A flurry of activity followed. MYCIN, a medical diagnosis system, the doctoral work of Edward Shortliffe, appeared in the early seventies. Its performance was rated highly by the Stanford Medical School, but it was never put to use mainly due to ethical and legal issues that could crop up. Another medical system was the *Internist* from the University of Pittsburgh. A system to help geologists’ prospects for minerals, *Prospector*, was developed by Richard Duda and Peter Hart. Buying a computer system was not an ‘off-

the-shelf' process as it is now, and needed considerable expertise. A system called R1, later named XCON, was built in CMU in 1978, to help users configure DAC VAX systems.

The main problem with the idea of Expert Systems was what is known as the *knowledge acquisition bottleneck*. Despite the elaborate interviewing protocols that researchers experimented with, domain experts were either unable or unwilling to articulate their knowledge in the form of rules. And like in other domains, the lure of deploying techniques to extract performance directly from delving into data was becoming more appealing. In the nineties, things began to change. The Internet was growing at a rapid pace. Automation was happening in all kinds of places. The problem of getting data was the least of the problems; making sense of it was.

By this time, we were at the cusp of the 21st century.

There were other strands of investigation in AI research that had been and were flourishing. Research areas like qualitative reasoning, non-monotonic reasoning, probabilistic reasoning, case based reasoning, constraint satisfaction, and data mining were evolving. Robots were becoming more and more capable, and robotic football was providing an exciting domain for integration of various ideas. The ageing population in many advanced countries is motivating research in robotic companions and care givers. John Laird had declared that the next level of computer games is the killer application for artificial intelligence. Search engines were ferreting out information from websites in far flung areas. Machines talking to each other across the world demanded advances in Ontology. Tim Berners-Lee had put forth the idea of the Semantic Web. NASA was deploying AI technology in deep space. *Kismet*⁸ at MIT was beginning to smile.

The last frontier is perhaps *Machine Learning*, the automatic acquisition of knowledge from data, which seems again a possibility, given the increasing ability of machines to crunch through large amounts of data. But not before, as Haugeland says in his book, *AI: The Very Idea*, before we have solved the knowledge representation problem.

As we have seen, the juggernaut of AI has been lurching along. Quite like the search methods AI algorithms embody, AI research has also been exploring various avenues in search of the keys to building an intelligent system. On the way, there have been some dead ends, but there have been equally, if not more, stories of achievement and progress. And on the way, many interesting and useful applications have been developed. Are these systems intelligent? Will we achieve artificial intelligence? Perhaps we can keep that question on the backburner for some more time. And, as Shakespeare said⁹,

"What's in a name? that which we call a rose

By any other name would smell as sweet;"

Nevertheless, the question of *what is* intelligence is an intriguing one. In the following section, we take a fundamental look at this question, keeping in mind that one of the goals of artificial intelligence research is

to implement programs that take us closer to achieving it.

1.3 What is Intelligence?

Everyone has an intuitive notion of intelligence. But at the same time, intelligence has eluded definitions. We often try to define it by certain functionalities, which one might associate with the competition to survive. The ability to perceive the world around us, to procure food for satisfying one's hunger, to pick a spot of shade to avoid the sun in the tropics, or a spot of warm sun in a cooler climate, to sense danger and flee, or fight if escape is not possible, are surely characteristics of intelligent behaviour. Who would ever call something intelligent if it kept standing in the sweltering sun, eschewing the shelter of a nearby tree? But that is perhaps not all there is to intelligence, for if it were then cats and deer and lions would be intelligent too. They *are*, in this sense and to this extent, intelligent. But intelligence, as we want to define or recognize, must include something more. There is the ability to reason, do arithmetic, and play chess. This of course is well within the capacity of the computer, but this is not sufficient to be called intelligent too. The ability to use language and learn from experiences must count as well. We know that animals learn from experience, and that certain species do communicate with some form of meaningful sounds. Other species do use signs and symbols to mark their territory and lay trails for others to follow. The question we ask then is do they do so consciously, or is it a part of their inherited genetic makeup?

Gradually, a picture emerges that intelligence may be (the effect of, or emerge from) a collection of different kinds of skills working together. It is as if a society of problem solvers inhabits and *makes up* our minds (see (Minsky, 1988)).

Goal oriented behaviour must be part of intelligence, and the deployment of skills is usually directed towards the achievement of certain goals one may have. Such goal directed behaviour may require collection and organization of large amounts of knowledge, which we believe computers should be good at. But we also believe that forgetting unimportant things is part of intelligent behaviour. Knowledge may be hard-wired, such as that of a weaver bird that builds the intricate nest, or the beaver that can build a dam; or it may be acquired through a process of learning, such as when human infants do. One might of course ask whether learning itself is hardwired, or is it an acquired skill. It is the ability to acquire knowledge in a specialized domain that we often feel is beyond the grasp of other animals. After all, who would not call a well-dressed cigar-smoking rabbit who can talk English and beat everyone at chess, intelligent? Then there are the higher order faculties that we associate with intelligence. One of them is creation, or even appreciation of fine art. Sometime in the mid-seventies of the last century, Harold Cohen created a program called *AARON* whose paintings have been

much admired (McCorduck, 1990), (Cohen, 1995). Classical music has also been the forte of higher forms of intelligence. And yet a program called EMI (Experiments in Musical Intelligence) written by David Cope (1992; 1996; 2005) composed music in the style of over a hundred composers which seasoned listeners could not differentiate from the original composer's style.¹⁰

Does that mean that if we could build a robot with arms and legs, teach it to run, drink lemonade, be courteous and speak English in addition to its abilities of manipulating very large numbers with ease, beating us at chess while painting a study in still life, composing an aria for an opera, diagnosing illnesses and prescribing medicines, and planning and cooking a dinner for two—then we would have succeeded in creating artificial intelligence? 'No', comes the answer this time from a section of philosophers. The computer is only a symbol manipulator. It does not *know* what it is doing. It does not have a sense of being in this world. It does not know, for example, that it is adding money to someone's account even when it goes through a sequence of steps that results in doing so. It does not know that it is playing chess, or that it should win, or that there is chess, or that *it* is there. It is just programmed.

All this has something to do with *self awareness*. 'The computer may be able to beat me at chess', they say, 'but it is I who am intelligent, not it. Because I know that I am'. Ironically, grappling with the question of human identity, René Descartes had said "*cogito ergo sum*" (Latin for "I think, therefore I am" originally asserted by him in French "*Je pense donc je suis*"). Descartes felt that the very fact that he was thinking (the statement "*I am, I exist*", or for that matter any other thought) was enough to establish his existence, even if there were "*a deceiver of supreme power and cunning who is deliberately and constantly deceiving*" him. Much earlier, Plato had said "... so that whenever we perceive, we are conscious that we perceive, and whenever we think, we are conscious that we think, and to be conscious that we are perceiving or thinking is to be conscious that we exist..."¹¹. In recent times, it seems that when Warren McCulloch, a pioneer in neural models, was embarking upon his education at the age of 19, he was asked by the Quaker philosopher Rufus Jones on what he wanted to do. McCulloch replied "*I have no idea; but there is one question I would like to answer: What is a number that a man may know it, and a man, that he may know a number?*" At which his elderly mentor smiled and said "*Friend, thee will be busy as long as thee lives.*" (McCulloch, 1960). There have been many more people who have grappled with the notion of the self and consciousness. The interested reader is referred to (Hofstadter and Dennett, 1981) and (Hofstadter, 2007) for some very insightful explorations.

So, while for humans we use the process of thinking as a means of arriving at the knowledge of our existence, for computers we adopt the contrapositive. They cannot think because they have no knowledge of their existence. I know that I am playing chess. I want to win. I know that I want to win... and so on. The philosophical objection to artificial

intelligence is that since computers are not capable of self awareness, being just programmed machines, they cannot be called intelligent. One suspects that such objections are made in anticipation of technical advances. Earlier, the challenges were posed at a technical level, one of the more celebrated ones being the one by the chess grandmaster David Levy in 1968, who wagered a bet of £1000 that a computer could not beat him in the next ten years. He did not lose his bet, but in 1989 lost to the program *Deep Thought*, and in 1997, a chess computer beat the world champion Gary Kasparov (Hsu, 2002). David Levy, meanwhile, won the 2009 Loebner Prize contest (see below) and now believes that robots will be able to interact with humans in many meaningful ways, including at an emotional and physical level (Levy, 2008).

The requirement of self awareness lays the trap of solipsism.¹² How am I to be aware of someone else's self awareness? Can I even say confidently of another human being that she is self aware as I am? We all agree that sugar is sweet and the sky is blue. But how do I know that the sensations of sweetness that sugar gives rise in me, are the *same* as in you? We may both say that the sky is blue. But is your experiencing of blue the same as mine? When you say that you are feeling happy, can I really understand what you *feel*? If we answer yes to some of these questions then on what basis do we do that? Is it based on our physical likeness? Or the common language that we speak? Or the similar goals we seem to have?

And yet there is another argument against this criterion of self awareness. If I can ascribe it to a fellow human being, can I deny it to other animals, and other life forms? Surely, a cheetah is aware that it is hungry when it embarks upon a hunt. Then if we associate self awareness with life and require it to be a necessary criterion for intelligence then in effect we would be denying the computer the property of being intelligent; not because it cannot do something that one would expect of an intelligent agent, but because it is artificial.

One would have observed that definitions of intelligence are usually human centric. This may be simply because as humans, we can see life only from our own perspective.

1.3.1 The Turing Test

Perhaps, in order to circumvent the debate on intelligence and get on with the business of building machines that are intelligent, Alan Turing, prescribed a test to determine whether a machine (computer program) is intelligent (Turing, 1950). Turing himself felt that the question "Can machines think?" was meaningless. Instead, he proposed what he called "The Imitation Game" played between a man (*A*), a woman (*B*) and an interrogator (*C*), who can communicate with each other only through text messages. The interrogator knows *A* and *B* only as *X* and *Y*, and her (or his) task is to correctly label *X* and *Y* with *A* and *B*, by means of asking

them questions. In the game, A tries to mislead the interrogator. Turing then asks what would happen if a machine replaces the man in the role of A? If the interrogator decides wrongly as often as when A was the man then we could say that the machine was intelligent. This test has since come to be known as the Turing Test, and one can visualize it today as being played in a chat room with the interrogator chatting with the two entities.

In the paper, Turing (1950) anticipates various objections and offers rebuttals for them. He also says, *"I believe that in about fifty years' time, it will be possible to program computers, with a storage capacity of about 10^9 , to make them play the imitation game so well that an average interrogator will not have more than 70 percent chance of making the right identification after five minutes of questioning. ... I believe that at the end of the century, the use of words and general educated opinion will have altered so much, that one will be able to speak of machines thinking without expecting to be contradicted."*

Obviously, the Turing Test lays emphasis on the program behaving like a human. This means that a program participating in the test should conceal its abilities, like for example, to multiply very large numbers, or search through large amounts of information, or produce all anagrams of a given word in a jiffy. At the same time, it must display the kind of knowledge and language humans use, and engage in small talk about the weather, food prices and football scores. Some people feel that computers will never be able to do that, while others feel that the test is too simple.

A chatterbot program called *ELIZA*, written by Joseph Weizenbaum (1966) in 1966, showed how human judgment is fallible. The most popular version of *ELIZA* is a program running the script *Doctor*, and can be found amongst other places in the *emacs* editor. Weizenbaum said that the program parodies "the responses of a nondirectional psychotherapist in an initial psychiatric interview". He intended the program to be an exploration of natural language processing, but it was often taken too seriously by people who interacted with it. This was because the core of the program was to turn around the user's phrases. If you were to type *"I'm feeling sad"*, it would respond with something like *"Why are you feeling sad ?"*. Very often, if it did not know how to manipulate the current input, would resort to general questions like *"Tell me more about your family"*. It seems his secretary at MIT thought the machine was a real therapist, and spent hours revealing her personal problems to the program, only to be deterred when Weizenbaum told her (who was outraged at this invasion of privacy) that he had access to the logs of all the conversations.

Somewhat disturbed by the seriousness with which people were taking *ELIZA*, Weizenbaum wrote another book (Weizenbaum, 1976) to debunk the aura around the program, and said that it is we humans who tend to anthropomorphize such programs.

Nevertheless, since 1991, an annual competition called the *Loebner*

Prize has been held, in which a set of judges interact with a set of chatterbots, programs that chat, to decide which one is the most humanlike. Though no program has been able to fool the judges yet, there has been an admirable amount of sophistication in the participating programs over the years. Unlike *ELIZA* which had no knowledge backing its conversations, the modern versions have access to databases and topical world information. The 2009 competition was won by David Levy's program named *Do-Much-More*, in which it responded to a question about its identity with¹³ "*Well, I am studying engineering. That ought to give you an idea about who I am, but I would sooner be working at the Cadbury's factory.*"

One well known objection to the Turing Test was put forward by John Searle (1980) who says that programs *like ELIZA* could pass the Turing Test without understanding the words they were using at all. One must keep in mind that *ELIZA* itself has no understanding of words, but it cannot also be said to have passed the test. Searle illustrated his argument, now known as the *Chinese Room* argument, by a thought experiment. He imagines himself in a room, acting like a computer, manipulating Chinese symbols according to some rules. People outside the room slide in some slips with Chinese symbols, and Searle passes back some Chinese symbols in return. Does this mean that he understands Chinese? As one can see, this is a kind of Turing Test, in which success is determined by meaningful conversation. Searle does not address the issue of what kind of a program or rules would enable him to converse meaningfully with his interlocutors, and perhaps therein lies the question.

1.3.2 Intelligent Decisions

Let us attempt to look at intelligence from a more fundamental perspective. Intelligence is obviously about making the right choices and doing the right thing. The perceptive reader would have observed by now that we are on slippery ground again here; defining intelligence in terms of equally vague notions. But let us nevertheless make an attempt.

Doing the right thing would refer to some goals that the agent has. One might say that the fundamental goal of life is life itself. Consequently, the goal of living things is to live; to maintain the body by eating. From this point onwards, the 'boundedness' of resources comes into the picture. Bodies have finite lifespans, and life resorts to procreation. The individual may perish, but the species must survive. Food is a finite resource, and apart from having to compete for it, there is also the danger of ending up as a meal for someone. Competition leads to evolution, and mixing up of genes via mating becomes an effective means of improvement. The right thing, therefore, for an *individual* of a species, is to do whatever propagates the species. The *homo sapiens* (us humans, that is) developed cognitive abilities and have "won" the war of

species promulgation. We have organized ourselves into heterogeneous societies that benefit from the multitude of diverse efforts. In that sense, humans have done the right thing.

An agent can do the right thing only when the agent knows the situation. Intelligence is perhaps most associated with the faculty of understanding; the ability to perceive the world around us and comprehend the meanings of signs and symbols, including language; the ability to grasp the situation. A self aware agent will be able to conceptualize itself in a situation, choose the goals to achieve and the actions to achieve those goals. Taking to one's heels on spying a tiger in one's vicinity may be an act of cowardice, but more importantly is an act of intelligence derived from understanding the situation. The intelligent student will observe the teacher's face go red with anger and refrain from throwing that piece of chalk at his neighbour. The enthusiastic baseball player, who is not a blockhead, will realize that the accumulating dark clouds imply impending rain and will not implore his friends to hang on.

Having evolved a mind in which one can see oneself in the world, we have a situation in which individual human beings have their *own* personalized goals. That is, our individual goals are not necessarily in tune with the goal of the tribe, or community, or even the species. The goals of the species are long term, whereas the individual goals may be of shorter term. Long term goals and short term goals can often be in conflict, and whether one's choices or actions are intelligent or not would depend upon the context they are evaluated in. Thus, rampant consumption and exploitation of our earth's resources may be intelligent in the context of a few lifespans, but stupid from a longer term perspective. Individual goals may themselves be over varying time frames. Enjoying junk food in front of the television set may serve an immediate goal, but may not be a great idea over the long term, specially if it becomes a habit. Again, one sees that actions may be intelligent with respect to short term goals, but stupid with respect to long term goals. This may happen at smaller and smaller time frame levels. Tripping an opponent who is charging towards your goal in a football match may be a "right thing" if it averts the goal, but wrong if you get a red card and are sent out, and the opponents anyway score of the resulting free kick.

The point we are trying to make is that actions or decisions can only be judged in the context of the goal. Intelligence (or wisdom, in the larger context) may often lie in the selection of the goal to pursue. Having selected goals, one must figure out the actions that will achieve those goals. There is considerable scope for intelligence here as well. As we shall see later in the book, goals and actions will form a layered hierarchy. Goals are addressed by plans of actions, and may themselves be part of higher level goals. For example, packing a bag may involve a plan, but itself may be part of a holiday plan. Our endeavour in building intelligent machines will start at the bottom of the goal hierarchy, and then move upwards. Our task will be to devise means to find the right actions,

at any given level.

A software program controlling the stability of an aircraft will choose the right actions to do so at one level, a chess program may do so at another level. A route planner that gives you instructions in a car, a diagnostic program that tells you what is wrong with your copier, a program that teaches you how to solve problems in algebra and geometry, a storytelling machine, a program that guides you step by step through a recipe, or a program that acts as an organized memory repository for you, or cross checks your logical reasoning—all of these will fall within the scope of our work. We will not address the issues of whether these programs are intelligent or not, specially in comparison to conscious articulate creatures like ourselves.

Intelligence, thus, may manifest itself in the choice of goals an agent may choose, and the means adopted by the agent to achieve those goals. It also involves the ability to understand the situation and reason about the consequences of one's actions. It involves learning from experiences and adapting one's behaviour in a dynamic environment. Intelligence may involve doing calculated trade-offs between different options. It also shows up in appreciating an intricate work of art. But most of all, it involves being able to make sense of the world around us, and reasoning about change in this world.

1.4 The Bottom Line

The distinctive features of intelligence can perhaps be summed up in one word—*imagination*.

We might go so far as to say that intelligence is the power of imagination. Perception is being able to imagine the world around us, but imagination also allows us to create new worlds, like fiction writers do for us. Planning by projecting the effects of our intended actions is imagining what would happen if we carried out those actions. Listening to a story is fascinating because of our ability to imagine the events in the narrative. Watching a real life athlete do a high jump is similar to watching her do it on television, because in both cases, it is our imagination that is triggered. Sometimes of course, our imagination goes out of sync with reality, and then we say we are deluded or hallucinating. The exception however proves the rule. It is our ability to *reconstruct* the world around us in our heads, to imagine, that is at the core of intelligent behaviour.

Imagination is concerned with creating *models* of our surroundings in our heads, and being able to *reason* with those models. Reasoning may involve analysis or it may involve simulating the happenings in some form. The models may be close to reality, mirroring the world around us, in which case, we may want to validate them with “ground reality”. Or they may be abstract like those of a mathematician. Different professions teach us to make different kinds of models, some of which need a considerable amount of imagination to validate. Albert Einstein's theory of

relativity needed sophisticated experiments in astronomy to validate. The astronomer works with models of stars and galaxies, and is able to devise experiments to validate theories¹⁴ of different kinds. A chemist works with models of atoms and the bonds between them that form molecules. Complex molecules, like the double strand of DNA, are best described by constructing physical three-dimensional models. Biologists work with models of cells and organs. More recently, these models have become more and more precise and mathematical. Particle physicists imagine the workings of matter at sub-atomic level, and make predictions that may need an experiment in a cyclotron with a circumference running into kilometres.

But all of us model the everyday world around us and learn to map incoming information into this model and make predictions about actions that we and others do. Thus, an intelligent agent, or one might say an *imaginative agent*, can be modelled as shown in Figure 1.2. In the figure, the humanlike head *represents* the agent, and the world around it is depicted by the outermost zig-zag lines. The shaded oval inside the agent represents the model of the world that the agent has. Observe that we have not represented it as the dotted lines mirroring the outside world because the model of the world the agent carries is not perfect, and may be an abstract model. The humanlike head in the oval represents the fact that the agent is self aware, and can think of itself as part of the model of the world it carries. Of course, a deeper level of awareness would require that the model of itself that it carries should contain a model of the world, with itself in it. The agent “knows” the world. It knows that it knows the world. It knows that it knows that it knows the world. As one can imagine, this process can go on indefinitely, and practical constraints will dictate how aware the agent is of itself.

The ability to imagine can thus be seen as being at the heart of intelligence.

One, somewhat extreme view of the universe, called *digital philosophy*, is that the world itself is a computation being run, that is, it can be seen as a model being executed. “*The school of philosophy called pancomputationalism claims that all the physical processes of nature are forms of computation or information processing at the most fundamental level of reality*”.¹⁵ One of its main proponents, Edward Fredkin, who also used the term *digital physics*¹⁶, said that in principle, at least, there is a program or a computer that simulates the world in real time. A natural consequence of this that is of interest to us, is that we can never build perfect models of the universe that can make predictions of events faster than they actually happen. This means that all models we devise must necessarily be abstract to some degree, and must lose on accuracy in the interest of speed.

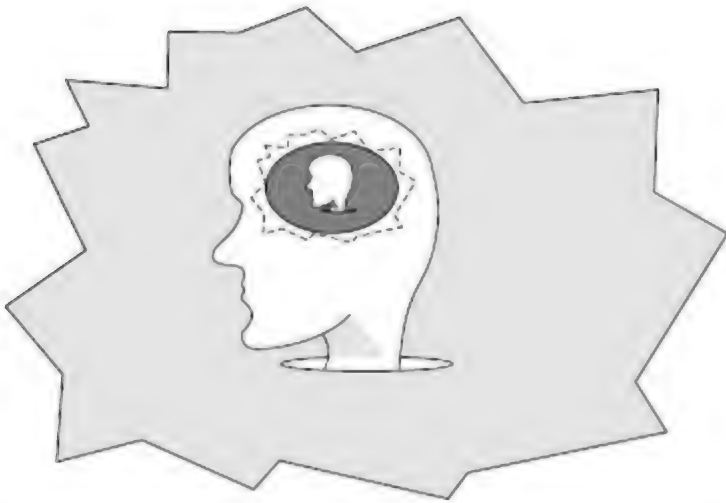


FIGURE 1.2 An intelligent agent in a world carries a model of the world in its “head”. The model may be an abstraction. A self aware agent would model itself in the world ‘model’. Deeper awareness may require that the agent represents (be aware of) itself modeling the world.

The reasoning that an agent does happens within the model that the agent has of the external world. The question that Descartes was trying to answer was about his own existence, and he said that the fact that he could think showed that he existed. Another question that has been raised is whether the world (really) out there is what we think it is? Given that everything that we can know is through our perception and cognition, how can we “*get outside our thoughts*” and know the “*real world*”. An extreme view known as *subjective idealism* or *immaterialism* put forward by George Berkeley (1685–1753), also known as Bishop Berkeley, says that there is no “*out there*” at all. His view on existence is “*esse is percipi*” (to be is to be perceived).¹⁷ Berkeley (1710) says “*objects of sense exist only when they are perceived; the trees therefore are in the garden ... no longer than while there is somebody by to perceive them.*” This has led to the well known philosophical riddle “If a tree falls in a forest and no one is around to hear it, does it make a sound?” In the ancient Indian philosophy school, *Advaita Vedanta*, the term *Maya* is used to refer to the limited mental and physical reality that our consciousness has got entangled in.¹⁸ It is *Maya* that creates the illusion of the physical universe to be seen by us, and may sometimes lead us to perceive a coiled piece of rope to be a snake.

Berkeley was opposing the views put forward by John Locke (1632–1704) that the *primary qualities* of a (material) object, such as its shape, motion and solidity, are inseparable from the object. Both belonged to the *Empiricist* school of philosophy that said that there existed an outside world in which our own ideas were rooted. But Berkeley argued that the outside world was also composed of ideas, and that ideas can only resemble ideas. He says that what we call as bodies are simply stable

collections of perceptions to which we give names like “tree”, “cloud”, and so on. If we believe that there are only ideas in the world (*idealism*) then we do not have to contend with the *mind-body dualism* that Descartes faced. More recently, Steve Grand (2001) has observed in his book that what we think is a cloud seen floating over a mountain is actually made up of moisture in the wind flowing over there. Our perception of a stationary cloud is only because condensation happens only in that region.

Even when we accept, like Locke, that there is a real world out there, we have to be careful about our perceptions about it. If we assume the atomic theory of matter then we can think of every physical object, including our own body, as made up of a large collection of atoms forming complex molecules, which in turn form cells, which in turn form organs, and so on. We tend to think of different clusters of atoms as individual objects and give them names. But biologists tell us that the bodies we had, say twenty years ago, were made up an entirely different set of atoms as compared to what we are made up of now. Yet, we think of our body as the same one we had as a child. Likewise, we know that a tree that we perceive in the forest has leaves that were not there last year, yet we think of it as the same tree. In his book, *Six Easy Pieces*, physicist Richard Feynman (1962) describes in detail how what we perceive as still water in a vessel is actually a humdrum of activity with atoms moving around, leaving the surface (evaporating) and reentering the liquid. We can extend this description to almost everything in the world around us. The fact that we tend to think of objects with well defined extents and surfaces is in fact the creation of our own minds.

An interesting anecdote that conveys our perception of objects is that of the Illinois farmer who possessed the (very same) axe used by Abraham Lincoln, where only the handle had been replaced twice and the head once. Also a work of *fiction*, the film *Matrix* (1999) by Andy Wachowski and Lana Wachowski, questions the nature of reality as perceived by humans, where the protagonist “*discovers that all life on Earth may be nothing more than an elaborate facade created by a malevolent cyber-intelligence*,”¹⁹ a throwback to the fears of Descartes.

The point we are trying to make here is the ability to imagine is what is needed for intelligence. An agent can exist intelligently if it is able to imagine the world it operates in. Intelligent systems can be built around the ability to represent and model the world, and reason about change in the model. The hypothesis called the *Physical Symbol System Hypothesis* stated by Alan Newell and Herbert Simon asserts that the ability to represent and manipulate symbolic structures is both necessary and sufficient to create intelligence (Newell and Simon, 1976). This means that it does not matter whether the agent is living or nonliving; carbon based biological form, an electromechanical robot or a virtual creature in cyberspace. All one needs is the ability to create symbolic representations and the ability to manipulate those representations. The degree of intelligence they demonstrate will depend upon the relevance

and richness of their representations and the effectiveness of their manipulation procedures.

1.5 Topics in AI

Philosophers have long dwelt over the mind and the human being that possesses that mind. They have asked questions about the universe and the mind that perceives the universe. Cognitive psychologists have studied how people acquire, process and store information. They investigate human memory, perception and decision making. Linguists have studied the use of language. Cognitive neuroscientists have investigated the biological systems that make up our brains, the seat of our minds. Economists have studied rational behaviour and decision making. Mathematicians have explored formal methods and the limits of logical reasoning.

The study of Artificial Intelligence can be seen to derive from all the above disciplines which can broadly be grouped under the name *Cognitive Science*. Except that the AI community is always focused on writing programs, whether it is to investigate and validate theories of human cognition, or it is to engineer systems for doing a particular task. The computer is a machine that allows us to represent and manipulate symbols explicitly, and symbols we believe, are the vehicles for ideas and concepts. And, as Patrick Henry Winston, professor at MIT, has observed, *“the computer is an ideal experimental subject, requires little care and feeding, is endlessly patient, and does not bite.”*

Computer programming has a quality that is quite different from most other activities. A team of programmers may pore over a procedure in the minutest detail, working out the required steps painstakingly; but when it is ready, the computer program runs in a jiffy, zipping through the steps that were determined by months of analysis. Written carefully in great detail, it can be invoked without care. Once a program is done, it can be executed and replicated as often as one wants. As more and more researchers write more and more programs, the ratchet of evolution will retain the best ones; and hopefully the different modules will evolve, and one day come together.

An intelligent agent would need a multitude of faculties. The agent should be able to sense the world around it, deliberate over its options, and act in the world. The different areas of study that are listed in Figure 1.3 are all part of the enterprise of building intelligent systems. One could think of these competencies as making up Minsky's *Society of the Mind*. The topics have been arranged in the figure so that sensing the world is to the left, and acting in the world is to the right. The world in this figure is the world inhabited by us humans, and we can think of the agent as a robot in the same world interacting with human beings. The topics inside the dashed circle are concerned with what we might call 'thinking'. The sense-deliberate-act cycle would describe the activity of an autonomous

agent in the world.

The topics on the left are concerned with sensing the world. As humans, we use the senses of sight, sound, touch, smell and taste. Of these, our current research focus is on the first three, though some work has been done in using smell as well (look up for example “artificial noses”). Visual perception is perhaps our most used sense, so much so that the word “seeing” has acquired a sense of “understanding” (when we say “I see”, for example). A variety of topics in computer science now address visual perception. Chief amongst those, depicted in the figure, are *Pattern Recognition* and *Computer Vision*, now areas of interest in their own right. The former, as the name suggests, studies the methods of recognizing patterns, while the latter deals with employing pattern recognition and other techniques to process still and moving images to identify objects and extract other kinds of information, and is often preceded by *image processing* which deals with the acquisition and preprocessing of image data.

Our sense of hearing gives us another window to the world. Sound can reach us when light often cannot, behind closed doors, deep in a forest, and even when we are sleeping. Humans have developed sound as the principal medium of conveying symbols, in the form of language. The written word came later. The disciplines of *speech recognition* and *natural language understanding* are concerned with the transduction of speech into language, and language into concepts respectively. Both may be concerned with dealing with patterns at different levels. Not shown in the figure is the faculty of creation and appreciation of music, which has been another manifestation of making sense of sounds. The sense of touch too becomes important when we talk of building robots. A tactile sensor would allow a metal being to shake a human hand gently, and could also be used to sense the surroundings in some situations.

The topics on the right are concerned with acting in the world. On the language side, we have *natural language generation* and *speech synthesis* that express concepts in natural language and convert the words in the language into sounds respectively. We can also have music generation and along with computer graphics and animation, one can imagine a system that will generate a soothing tale at bedtime for our children, help us navigate through a new town, describe incidents from far flung planets, or give us a glimpse of tomorrow’s weather in our neighbourhood.

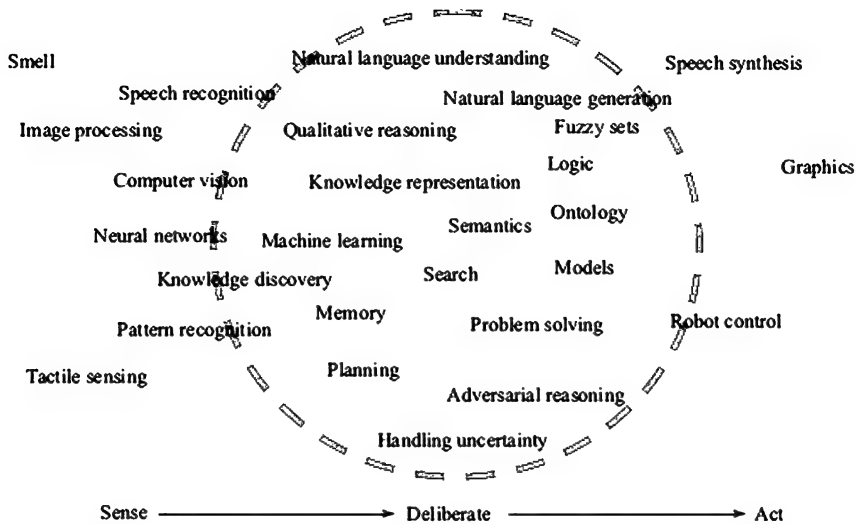


FIGURE 1.3 A sampling of the "society of the mind". Different kinds of reasoning methods would be required by an intelligent mind, as it goes through a sense-deliberate-act cycle. The dashed circle represents symbolic reasoning in artificial intelligence.

But the core of intelligence lies in the centre, dealing with knowledge, language and reasoning. These are the faculties we believe "lower" forms of life do not possess. These are the means that make imagination possible: the ability to conceptualize and create models of the world around us; to categorize things and create memories of our experiences; to reason with what we know and make predictions by projecting actions into the future; to handle uncertainty and incomplete knowledge; to take into account the options of adversaries and collaborators, and to formulate problems and search for their solutions. These are the topics that will form the subject matter of our text. These are the processes that can be modelled as operating on symbol systems.

One could adopt the view that the agent and the world it interacts with are entirely symbolic in nature. In practice, however, we often build systems that treat the external world as analog, consisting of continuous signals. Thus, we talk of speech signals which we digitize to create digital sequences. One can imagine that the core symbolic reasoning system has a shell of processes around it that converts signals to symbols and vice versa. It has been empirically observed that neural networks and fuzzy reasoning systems can serve as a bridge between symbolic representations and analog signals, and vice versa. For example, it is much more effective to train a neural network to recognize handwritten characters. It is not easy for us to define rules that capture the myriad variations in human writing; but feed a large number of examples to a neural network, and it soon learns to the symbols that make up words from the (digitized) squiggles one makes with pen and paper. Likewise, a fuzzy controller would take a symbolic command like "warm" and generate an appropriate signal to a heater in some contraption. The

physical systems that interact with the world, for example a microphone to capture sound, or an accelerator pedal in an autonomous vehicle, may operate entirely with signals. This way of looking at things is depicted in Figure 1.4 in which the core symbolic system is shown as having a neuro-fuzzy shell around it. The neuro-fuzzy shell is instrumental in mediating between the symbolic reasoning system and the external continuous world.

The subject matter of this textbook is the inner core of symbolic reasoning, and is sometimes referred to as *Classical AI*.

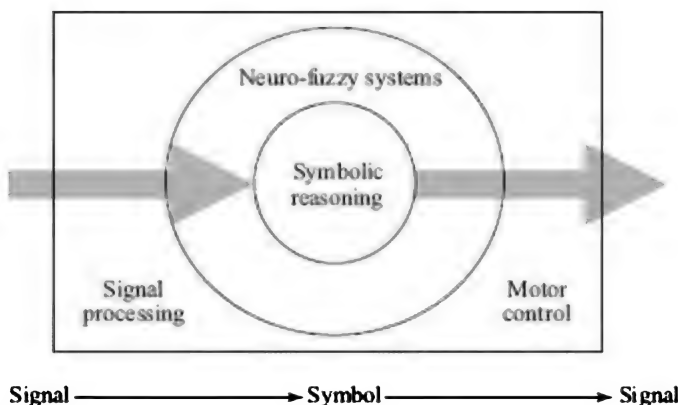


FIGURE 1.4 A model of a cognitive agent. The innermost circle represents Classical AI that is concerned with symbolic reasoning. This is encapsulated in neuro-fuzzy systems that produce and consume the symbols. The outermost layer deals with the external world directly, processing signals and producing motor activity.

Incidentally, one must realize that the distinction between analog and digital is entirely of our own making, and mostly reflects the nature of our devices. The earliest radios were made up of vacuum tubes, as were the earliest computers until the invention of the diode. Likewise, our telephones, television sets and cameras started off as analog devices and are now digital in nature. We can think of it as digital systems going more and more into the analog world. But we can also think of the analog representation as a way of looking at a world that is entirely digital, as Ed Fredkin would have us think. After all, we may think of continuously varying voltage and current, but the electrical signal is ultimately made up of electrons, which we do tend to think of as distinct “particles”. Similarly, we tend to think of flowing water as continuous material but we do know that it is a large collection of individual molecules obeying the laws of physics. Signal processing is becoming more and more digital signal processing. If we simulate a neural network then the weight of its synapses is represented by a number of finite precision in a computer. In the end, anything that we do on a digital computer has to be symbolic at the deepest level, because the digital machine can in the end distinguish between only two kinds of symbols—0 and 1.

Nevertheless, we shall be concerned with only those symbolic

systems in which the symbols mean something to us. These will include numerals to represent numbers, or words like “apple” and “love”, or even a variable routinely named X in a program. We will not deal with neural networks and similar systems in which we cannot interpret weights of edges, and what a node represents. We will be concerned, however, with networks of nodes, semantic nets, in which each node will represent something meaningful to us. The following section describes the contents of this book with some description that would allow one to eschew the linear narrative, and jump directly to some topic of interest.

A Brief Look at the Chapters

The study of *Artificial Intelligence* is concerned not so much with defining intelligence, but the study of the different kinds of reasoning processes that would come together to create an intelligent system. And an intelligent system would be one that can represent its domain of activity, perceive the state of the domain, and reason in a manner to achieve its goals. This reasoning process we refer to as *problem solving*.

There are two basic approaches to solving problems. One is to create an appropriate representation of the domain, and *project* the consequences of one's decisions into the *future* by some kind of simulation. One can investigate different possible courses of actions and choose the most promising one. We can say that this approach is a *first principles approach*, in which a problem is solved anew when it presents itself. However, as we will see in the book, such methods incur a high computational cost since the number of combinations explodes exponentially in many domains. The second approach is to delve into the agent's *memory*, and retrieve similar problems that were solved in the *past*. Both approaches require knowledge representation to model the domain, the entities and the relations between them. In addition, systems that rely on memory also need to represent episodes from the past.

In the first half of the book, we look at problem solving using search. The given situation and the desired situation are mapped on to a search space. We assume that the solver has access to “moves”, using which the solver can change the situation a little bit. Each application of a move transforms the situation, taking the solver to a new state. Problem solving using search involves exploring, by a process of trial and error; and finding the sequence of moves that will take the solver to the desired state.

Chapter 2 introduces the basic procedures for exploring the search space, and discusses the complexity of search algorithms.

Chapter 3 introduces heuristic functions and heuristic search. It also introduces greedy search methods.

Chapter 4 looks at randomized approaches to search and discusses some popular optimization techniques. We also look at emergent systems.

Chapter 5 extends the heuristic search algorithms of Chapter 3 to find

optimal solutions. It also describes variations that have a lower space requirement at the expense of increased time complexity.

Chapter 6 moves away from state space search and looks at problem decomposition and rule-based systems.

Chapter 7 looks at automated planning methods in which the moves are described as planning operators. The basic planning methods, including plan space planning, are covered.

Chapter 8 is devoted to game playing in which the problem solver has to contend with the moves of an adversary. We focus on board games like chess.

Chapter 9 introduces the alternative formulation of constraint satisfaction problems. We study the algorithms for constraint propagation and look ahead and look back methods in search.

Chapter 10 discusses the advanced planning algorithms that were developed in the last decade of the twentieth century. It also looks at richer planning domains in which actions have durations, along with some real world applications.

Solving problems from first principles has its advantages. The principal one being that one is solving the given problem. However, this approach is fraught with computational pitfalls. Every trial-and-error search method has to contend with a danger lurking in the deep. That deeper one searches the greater number of possibilities one has to consider, not unlike the monster *Hydra* that Hercules has to fight (in Greek mythology). For every head Hercules severed, several more appeared. A similar thing happens in search. Every failed possibility gives rise to several new ones. We will call our monster *CombEx*, for combinatorial explosion, the bane of every search method.

And yet we can fight CombEx. The weapon we deploy successfully in our daily lives is knowledge; knowledge about how to do things, knowledge about what the world is like, and knowledge gleaned from our experiences and that of others.

The second half of the book is concerned with knowledge representation and reasoning. This is in contrast to the first half that ignored how the world was represented. In the second half, we explore ways to represent different kinds of knowledge and study algorithms to exploit that knowledge.

Chapter 11 introduces the issues dealt with in the second half.

Chapter 12 deals with first order logic and theorem proving methods.

Chapter 13 is concerned with how we express everyday facts in first order logic. We also look at the process of reification that allows us to extend the logic to talk about events and actions.

Chapter 14 introduces the notion of knowledge structures. We look at organizing knowledge into packages and hierarchies, and look at notions of inheritance.

Chapter 15 is concerned with problem solving by remembering and reusing experiences. The simple process of storing experiences and recalling relevant ones has proven to very effective.

Chapter 16 focuses on natural language processing. We study the techniques needed to process a sentence in language and the task of disambiguation that crops us. We also look at text processing applications like information retrieval and machine translation.

Chapter 17 looks at various methods of dealing with uncertainty. We begin with extending logics to handle incomplete knowledge, look at qualitative approaches to reasoning, reasoning with assumptions, reasoning with probability.

Chapter 18 is the last chapter dealing with machine learning, an area that will determine how machines can acquire more knowledge by themselves. We begin with concept learning and decision trees, and follow up with well established techniques like *Naïve Bayes Classifier* and *Hidden Markov Models*, and approaches to clustering. We also look at learning in Artificial Neural Networks.

A word about the algorithms presented in this book. We begin by describing them in considerable detail, but later when we build variations, we make our descriptions a little bit more abstract, focusing on the principal ways the new algorithms differ from the ones described earlier. Every now and then, however, we find the need to describe an algorithm in greater detail.

Even though our long term goal is to build a fully integrated intelligent system, the journey can be productive as well. During the last fifty years, AI researchers have built many useful systems that use only a few reasoning methods individually. These are systems we may not want to label as being 'intelligent', but they do employ some of the aspects of intelligence we have discussed. These systems include a chemist's expert assistant, a system that helps explore the earth for minerals, systems that advise a patient on whether they need to rush to a hospital or not, tutoring systems that teach children math, systems that can prove theorems, systems that can create an organizational memory, system that help plan and book your travel, systems that can ferret out useful information from the World Wide Web, systems that would recommend a movie and a restaurant to you, systems that would respond to verbal commands of the user, systems that would monitor the activity of the aged and remind them of activities they need to do, systems that analyse large amounts of data and discover useful knowledge, systems that help translate from one language to another, systems that can plan the movement of rover on another planet, or an autonomous submarine in our own seas, systems that generate weather forecasts in a coherent manner, systems that help monitor the progress of neonatal babies and produce reports for anxious parents, and many others including, not least of all, systems that play with us and entertain us.

The development of all these systems has taken place independently by different groups in different parts of the world. Many of these systems demonstrate the capability that would be part of a general intelligent system. It is only to be expected that with further advances, more and more capabilities will be integrated into single systems, and perhaps one

day, we will have intelligent companions who behave like friends to us, giving us advice and telling us jokes, but who are not embodied in flesh and bones like us.



Points to Ponder

1. We cannot represent the full world.
We cannot sense the full world.
We cannot (always) act perfectly.
We cannot predict the consequences of proposed actions perfectly.
How then does intelligence arise?
2. Our brains are finite, made up of tens of billions of neurons. The number of different sentences we can construct in (say) the English language is much larger. When we are faced with a new sentence we have never seen before, how do we make sense of it?

¹ We must admit though that the situation is not ideal, being far better for some of us than for others.

² See also <http://www.fourmilab.ch/babbage/contents.html>

³ Hobbes in the comic strip *Calvin & Hobbes* was named after him (Waterson, 1995).

⁴ Named after the fictional computer of the same name in Douglas Adam's celebrated novel, *The Hitchhiker's Guide to the Galaxy*. The computer was created by a pan-dimensional, hyper-intelligent race of beings to come up with the *Answer to The Ultimate Question of Life, the Universe, and Everything*.

⁵ See <http://www.robothalloffame.org/>

⁶ The name derives from a 1942 play of the name "ETAOIN SHRDLU" by Frederic Brown about an artificially intelligent Linotype machine. The title itself was based on the order of letter keys on the Linotype machine.

See <http://hci.stanford.edu/~winograd/shrdlu/name.html>

⁷ See <http://wordnet.princeton.edu/>

⁸ <http://www.ai.mit.edu/projects/humanoid-robotics-group/kismet/kismet.html>

⁹ In *Romeo and Juliet*

¹⁰ See for example "Sounds like Bach" by Douglas Hofstadter, at <http://www.unc.edu/~mumukshu/gandhi/gandhi/hofstadter.htm> (accessed on August 20, 2010).

¹¹ See http://en.wikipedia.org/wiki/Cogito_ergo_sum

¹² See <http://en.wikipedia.org/wiki/Solipsism>

¹³ From <http://www.worldsbestchatbot.com>

- ¹⁴ We use the term 'model' in the sense of the word theory, something that stands for the real thing and needs to be validated. Logicians also use word 'theory' for what we mean here by a model, and model for a domain and a mapping that satisfies the theory.
- ¹⁵ See http://en.wikipedia.org/wiki/Digital_philosophy and http://en.wikipedia.org/wiki/Ed_Fredkin
- ¹⁶ See http://en.wikipedia.org/wiki/Digital_physics
- ¹⁷ See http://en.wikipedia.org/wiki/George_Berkeley
- ¹⁸ See [http://en.wikipedia.org/wiki/Maya_\(illusion\)](http://en.wikipedia.org/wiki/Maya_(illusion)) for a detailed description.
- ¹⁹ Plot summary by Jake Gittes at <http://www.imdb.com/title/tt0133093/plotsummary>

State Space Search

Chapter 2

Problem solving basically involves doing the right thing at the right time. Given a problem to solve the task is to select the right moves that would lead to the solution.

Consider the situation in a football game you are playing. You are shown as player *P* in Figure 2.1 below. You have the ball with you and are running towards the opponent's goal. There are some other players in the vicinity also shown in the figure along with the directions of their movement. You have a problem to solve. What should you do next? What move should you make? What is the "intelligent" thing to do?

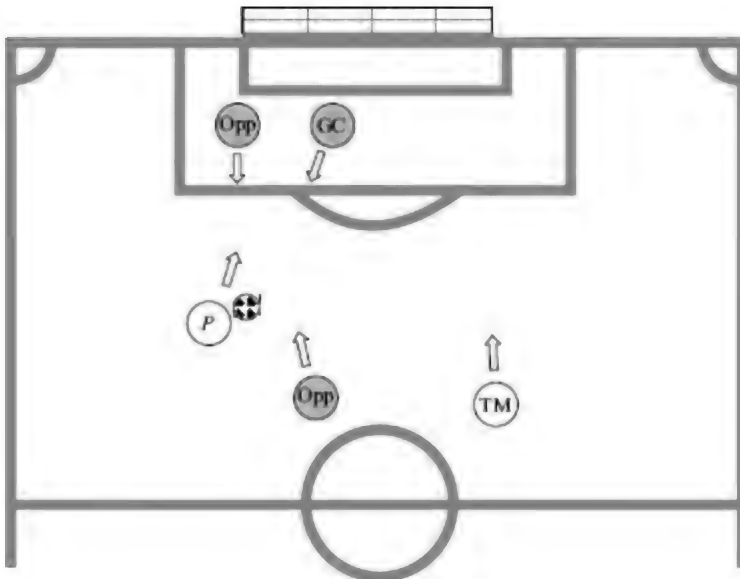


FIGURE 2.1 A problem to solve on the football field. What should player *P* do next? GC is the opponent goalkeeper, Opp are opponents, and TM is teammate. Arrows show the direction of each player's movement.

To solve such a problem on a computer, one must first create a representation of the domain; in this case, a football game being the problem, the decision to make next, the set of alternatives available, and the moves one can make. The above example is a problem in a multi-agent scenario. There are many players and the outcome depends on

their moves too. We will begin, in the initial chapters, with a much simpler scenario, in which there is only one player who can make changes in the situation. We will look at algorithms to analyse the complete problem and find a solution. In later chapters, we will explore ways to deal with multi-agent scenarios as well.

We will also use the term *agent* to refer to the problem solver. The problem solver, or the agent, operates on a *representation* of the *space* in which the problem has to be solved and also a representation of the *moves* or decisions that the agent has to choose from to solve the problem.

Our endeavour will be to write the algorithms in a domain-independent manner. Then the algorithms will be general purpose in nature and could be adapted to solve problems in different domains as depicted in Figure 2.2. A user would only have to implement the domain description and call the domain specific functions from the general program.

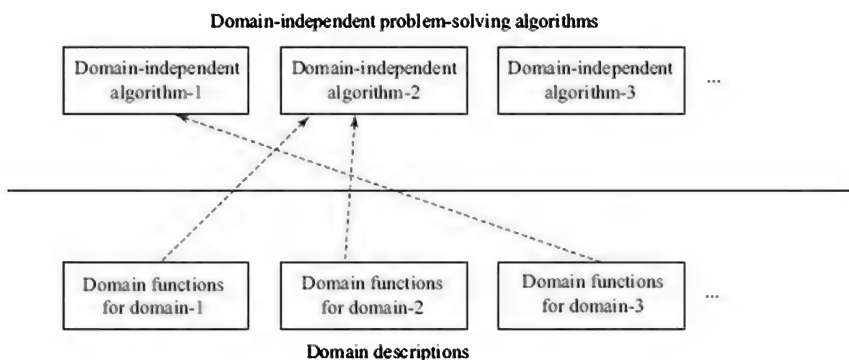


FIGURE 2.2 The goal is to develop general problem solving algorithms in a domain-independent form. When a problem is to be solved in a domain then the user needs to create a domain description and plug it in a suitable problem solving algorithm.

We model the problem solving process as traversing a *state space*. The state space is a space in which each element is a state. A state is a description of the world in which the problem solver operates. The given situation is described by a state called the *START* state. The desired or the goal situation is described by one or more *GOAL* states. In any given state, an action or a decision by the agent changes something and the agent makes a *move* to a new state. The task is to make a sequence of moves, such that the agent ends up being in a goal state.

The set of choices available to us *implicitly define* the space in which the decision making process operates. We do not assume that the entire state space is represented in some data structures. Instead, only the states actually *generated* by the agent exist explicitly. The unseen states are only implicit. In this implicit space, we visualize the problem as follows. Initially, we are in some given state, or the *START* state. We desire to be in some state that we will call the *GOAL* state, as shown in Figure 2.3. The desired state may be described completely, identifying

the state, or it could be described partially by some desirable properties; in which case there may be more than one goal state satisfying the properties. The given state being the current state is described completely.

This transformation from the start state to the goal state is to be made by a sequence of moves that are available to us in the domain of problem solving. In the beginning, the search algorithm (or agent) can only “see” the start state. It has access to some move generation operators that determine which states are reachable in one step (see Figure 2.6). The algorithm has to choose one of these moves. Each move applied to a given state transforms it into another state. Our task is to find those sequences of moves that will transform our current (*START*) state into the desired (*GOAL*) state. The solution is depicted in Figure 2.4.

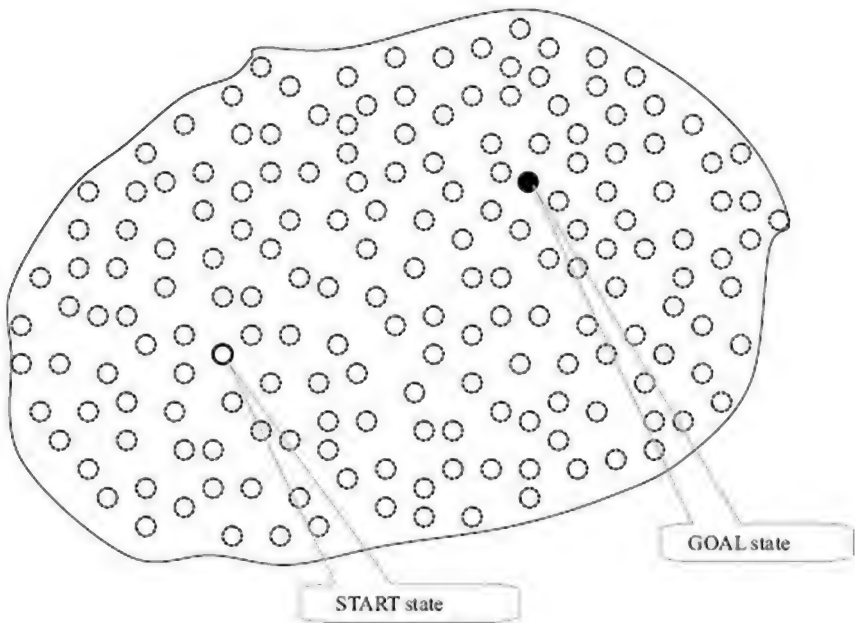


FIGURE 2.3 The state space.

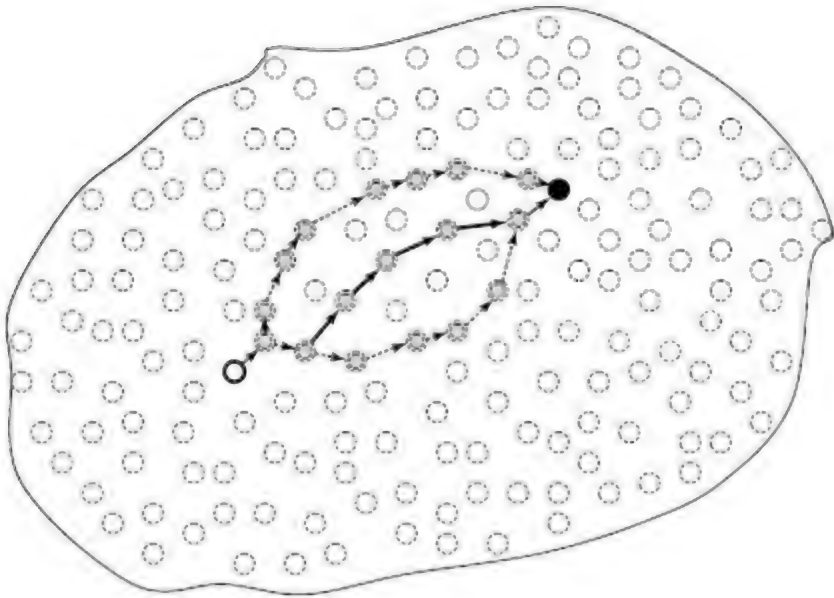


FIGURE 2.4 The solution is a sequence of moves that end in the GOAL state. There may be more than one solution to a problem. The figure shows one solution with thick arrows, and two alternatives with dotted arrows.

2.1 Generate and Test

Our basic approach will be to search the state space looking for the solution. The high level search algorithm has two components; one, to generate a candidate from the state space, and two, to test whether the candidate generated is the solution. The high level algorithm is given below in Figure 2.5.

```

Generate And Test()
1  while more candidates exist
2    do  Generate a candidate
3        Test  whether it is a solution
4  return FAILURE

```

FIGURE 2.5 A high level search algorithm.

The rest of the chapter will be devoted to refining the above algorithm. We assume that the problem domain has functions defined that allow us to operate in the domain. At the moment, we need two functions to be defined on the domain. They are the following:

moveGen(State) Takes a state as input and returns a set of states that are reachable in one step from the input state, as shown in Figure 2.6.

We call the set of states as *successors* or *children* of the input state. The input state is the parent of the children.

goalTest(State) Returns *true* if the input state is the goal state and *false* otherwise.

goalTest(State, Goal) Returns *true* if *State* matches *Goal*, and *false* otherwise.

Observe that we may have either of the above goal-test functions. The former is used when the goal is described by some properties that are checked by the function. The latter takes an explicit goal state and uses that to compare with the candidate state.

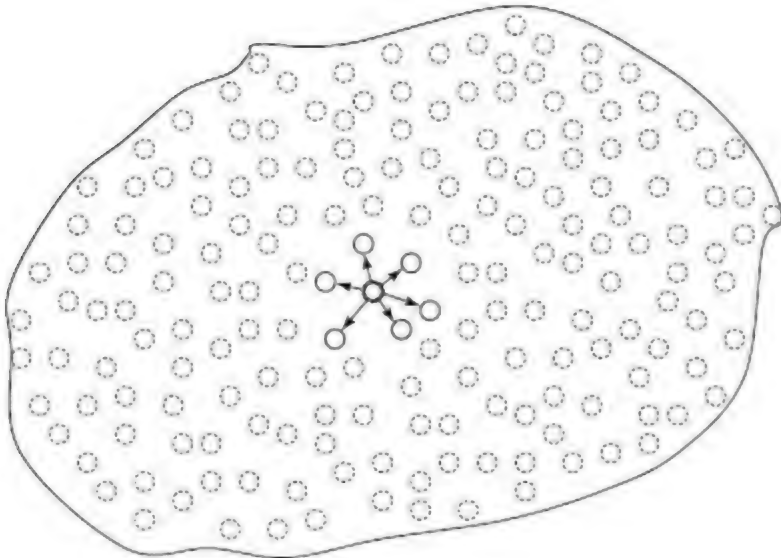


FIGURE 2.6 The moveGen function returns a set of states that are reachable in one move from a given state.

Our search algorithms should be able to operate in any domain for which the above functions are provided. We view the set of states as 'nodes' in the state space, and the set of moves as the edges connecting these nodes. Thus, our view of the state space will be that of a graph that is defined *implicitly* by the domain function moveGen. Figure 2.7 depicts the part of the state space generated and explored by the *Generate&Test* algorithm. The generated space is known as the *search tree* generated by the search program.

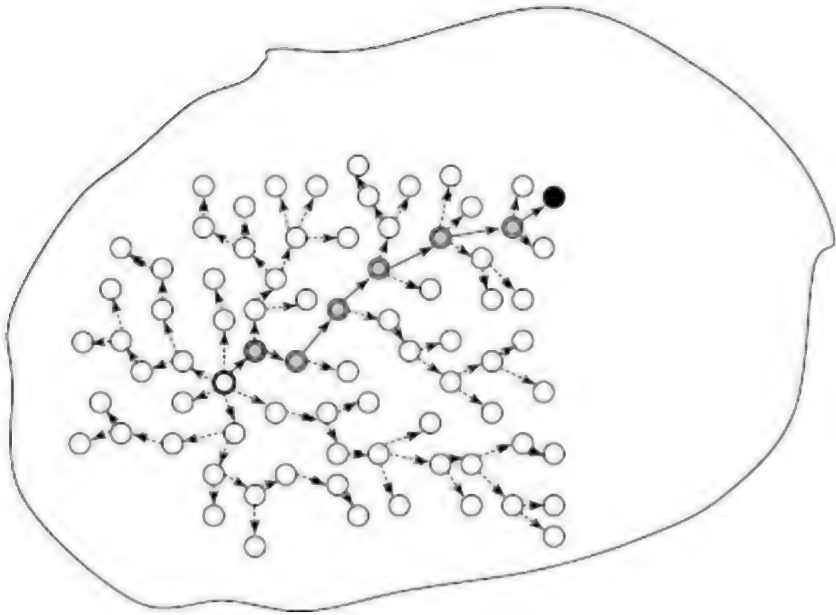


FIGURE 2.7 The nodes visited by a search algorithm, form a search tree shown by empty circles and dotted arrows. The solution found is shown with shaded nodes and solid arrows.

Before looking at the algorithms in detail, let us look at a couple of problem examples. While many real world problems can indeed be posed as search problems, we prefer simpler problems often posed as puzzles. Small, well defined problems are easy to visualize and easy to implement, and serve to illustrate the algorithms. Real problems on the other hand would be complex to represent, and divert from the understanding of search methods. Nevertheless, the reader is encouraged to pose real world problems as state space search problems. A few suggestions are given in the exercises.

Common sorts of puzzles are river-crossing puzzles. In these puzzles, a group of entities need to cross the river, but all of them cannot cross at the same time due to the limited capacity of the boat or the bridge. In addition, there may be constraints that restrict the combination of entities that can stay in one place.

One example of a river-crossing puzzle is the missionaries and cannibals problem stated as follows. There are three missionaries and three cannibals who want to cross the river. There is a boat with a capacity to seat two. The constraint is that if on any bank the cannibals outnumber the missionaries, they will eat them¹. How do all the six people get across safely? In the Text Box 2.1, we look at another river-crossing problem, and also design the domain functions required.

Box 2.1: The Man, Lion, Goat, Cabbage Problem

A man (M) needs to transport a lion (L), a goat (G), and a cabbage (C) across a river. He has a boat (B) in which he can take only one of them at a time. It is only his presence that prevents the lion from eating the goat, and the goat from eating the cabbage. He can neither leave the goat alone with the lion, nor the cabbage with the goat. How can he take them all across the river?

The state representation and move generation functions are interrelated. Let us say that we represent the state as a list of two lists, one for each bank of the river, say the left bank L and the right bank R. In each list, we name the entities on that bank.

```
| start state S = ((M G L C B) ())
| goal state G = (() (M G L C B))
```

The move generation could first generate all possible moves and then filter out illegal moves. For example, from the start state, the following states are reachable:

```
| ((G L C) (M B)), ((L C) (M G B)), ((G C) (M L B)), and ((G L) (M C B))
```

Of these, only the second state ((L C) (M G B)) is a legal state. Notice that the moveGen has transferred some elements from the first list to the second. In the next move it will need to transfer elements in the opposite direction. Also, there is redundant information in the representation. When one knows what is on the left bank, one also knows what is on the right bank. Therefore, one of the lists could be removed. To solve the problem of alternating directions, we could choose a representation that lists the entities on the bank where the boat is, and also which bank it is on, as shown below.

```
| start state S = (M L G C Left)
| goal state G = (M L G C Right)
```

where Left denotes that the boat is on the left bank and Right on the right bank. Note that M is redundant, because the man is where the boat is, and could have been removed.

The *moveGen(N)* function could be as follows.

```
| Initialize set of successors C to empty set.
| Add M to the complement of given state N to get new state S.
| If given state has Left, then add Right to S, else add Left.
| If legal(S) then add S to set of successors C.
| For each other-entity E in N
|     make a copy S' of S,
|     add E to S',
|     If legal (S'), then add S' to C.
| Return (C).
```

The complement of a state is with respect to the set {M L G C}.

The function “legal (state:S)” will return “no” or “false” if either both G and C or both L and G are missing from S, otherwise it will return “yes” or “true”.

The following figure shows the well known *Eight Puzzle* used extensively in one of the earliest textbooks on artificial intelligence (Nilsson, 1971). The goal is to find a path to a desired arrangement of tiles, for example as shown in Figure 2.8.

The idea in *weak methods* or general methods in artificial intelligence (AI) is that we develop a general algorithm that can be applied to many domains. In the above examples, we have illustrated how problems can be posed as search problems. Let us return to the design of the search algorithms. The following refinement of *Generate&Test* falls in the category of forward state space search. Here we begin with the *START* state and search until we reach the *GOAL* state.

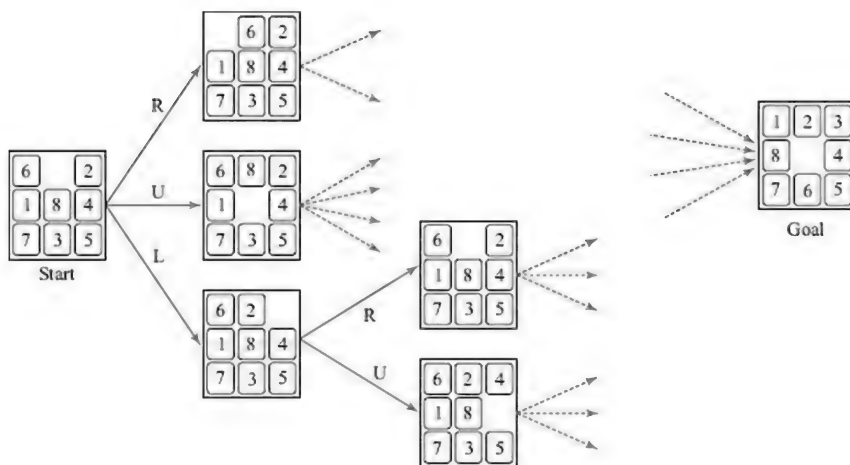


FIGURE 2.8 The *Eight Puzzle* consists of eight tiles on a 3×3 grid. A tile can slide into an adjacent location if it is empty. A move is labelled R if a tile moves right, and likewise for up (U), down (D) and left (L).

2.2 Simple Search 1

We assume a bag data structure called *OPEN* to store the candidates that we have generated. The algorithm is given below.

```

SimpleSearch1()
1 open ← {start}
2 while open is not empty
3   do pick some node n from open
4     open ← open \ {n}
5     if GoalTest(n) = TRUE
6       then return n
7     else open ← open ∪ MoveGen(n)
8 return FAILURE

```

FIGURE 2.9 Algorithm *SimpleSearch1*.

Box 2.2: Hercules, Hydra and CombEx

In Greek mythology, one of the monsters that Hercules found himself fighting was Hydra. Hydra was not an easy monster to deal with. For every head that Hercules chopped off, many more grew in its place. The problem faced by our search program is not unlike the one faced by Hercules. For every node that Simple-Search-1 picks to inspect, many more are added in the bag.

In mathematical terms, the search tree generated by the program grows exponentially. And yet, as Hercules demonstrated in his fight with Hydra, the monster can eventually be overcome.

Almost all the research we see in artificial intelligence can be seen as a battle against CombEx, the monster that makes our problems grow with a 'COMBINatorial Explosion'.

Our battle with CombEx will require two abilities. The first will be the basic skills to navigate and explore the search space. This is akin to learning how to handle the sword. But as we will see, this will not be enough. We will also need to know where to strike. Without knowledge to guide the exploration, search will be hopelessly outnumbered by the exploding possibilities.

The first is necessary because CombEx will appear in new forms as we strive to solve new problems. But as we wage the battle, we must be able to learn the strengths and weaknesses of CombEx in every domain. We must acquire knowledge both through our own experience and the accumulated experience of other problem solvers.

Let us look at a small, synthetic problem to illustrate our search algorithms. Figure 2.10 depicts the state space graph for a tiny search problem.

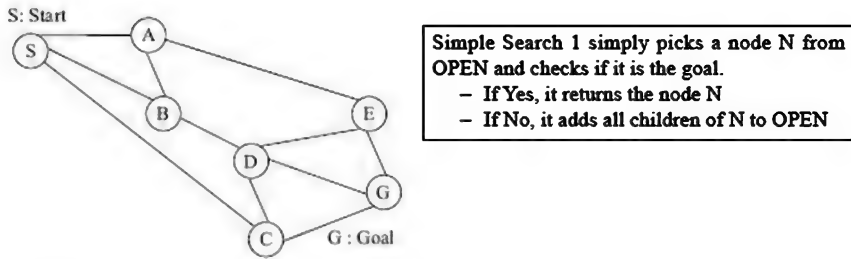


FIGURE 2.10 A tiny search problem.

There are two problems with the above program. The first is that the program could go into an infinite loop. This would happen when the state space is a graph with loops in which it would be possible to come back to the same state. In the *Eight Puzzle* described above, the simplest loop would be when a tile is moved back and forth endlessly. For the tiny search problem of Figure 2.10, a possible trace of the search algorithm is shown below in Figure 2.11, in which the same nodes are visited again and again.

Mazes are a very good example where infinite loops lurk dangerously. They also illustrate the nature of difficulty in the search. See the accompanying

```

(S)
(ABC)
(SACD)
(ABCACD)
(SACACD) ...

```

FIGURE 2.11 Simple search may go into loops. One possible evolution of OPEN is shown above. The node picked at each stage is underlined.

Box 2.3: The Problem of Visibility in Search

Anyone who has gone trekking in the mountains would have experienced the following. You are at a peak looking down towards a village in the valley. You can clearly chart a path through the woods on the way down, past those set of boulders, back into the woods, and then following a path along the sunflower fields. But when you start descending and enter the woods, the perspective and clarity you had from the top curiously vanishes.

A similar problem is faced by people navigating a network of roads. It is all very clear when you are poring over a map. But down at the next intersection, it becomes a problem which road to take.

The search algorithms we are trying to devise do not have the benefit of perspective and global viewpoints. They can only see a node in the search space, and the options available at that node. The situation is like when you are in a maze. You can see a number

of options at every junction, but you do not have a clue about the choice to make. This is illustrated in Figure 2.12.

In the next chapter, we will investigate methods to take advantage of clues obtained from the domain. If for example, one of the paths in the maze has a little more light than the others, it could be leading to an exit.

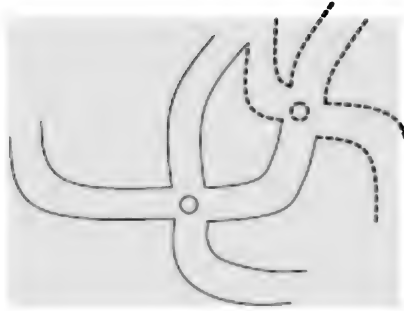


FIGURE 2.12 In a maze, one can only see the immediate options. Only when you choose one of them do the further options reveal themselves.

Box 2.3 for a note on the limited visibility in search. The key to not getting lost endlessly in a maze is to mark the places where one has been. We can follow a similar approach by maintaining a list of seen nodes. Such a list has traditionally been called *CLOSED*. It contains the list of states we have tested, and should not visit again. The algorithm *Simple Search 2* (SS-2) in Figure 2.13 incorporates this check. It does not add any seen nodes to *OPEN* again. To prune the search tree further, one could also specify that it does not add any successors that are already on *OPEN* as well. This would lead to a smaller search tree in which each node occurs exactly once.

```
SimpleSearch2()
1 open ← {start}
2 closed ← {}
3 while open is not empty
4   do Pick some node n from open
5     open ← open \ {n}
6     closed ← closed ∪ {n}
7     if GoalTest(n) = TRUE
8       then return n
9     else open ← open ∪ {MoveGen(n) \ closed}
10 return FAILURE
```

FIGURE 2.13 Algorithm *SimpleSearch2*.

The search tree generated by one possible execution of the algorithm is shown below. The *CLOSED* list is depicted by shaded nodes and the

OPEN list by dotted circles.

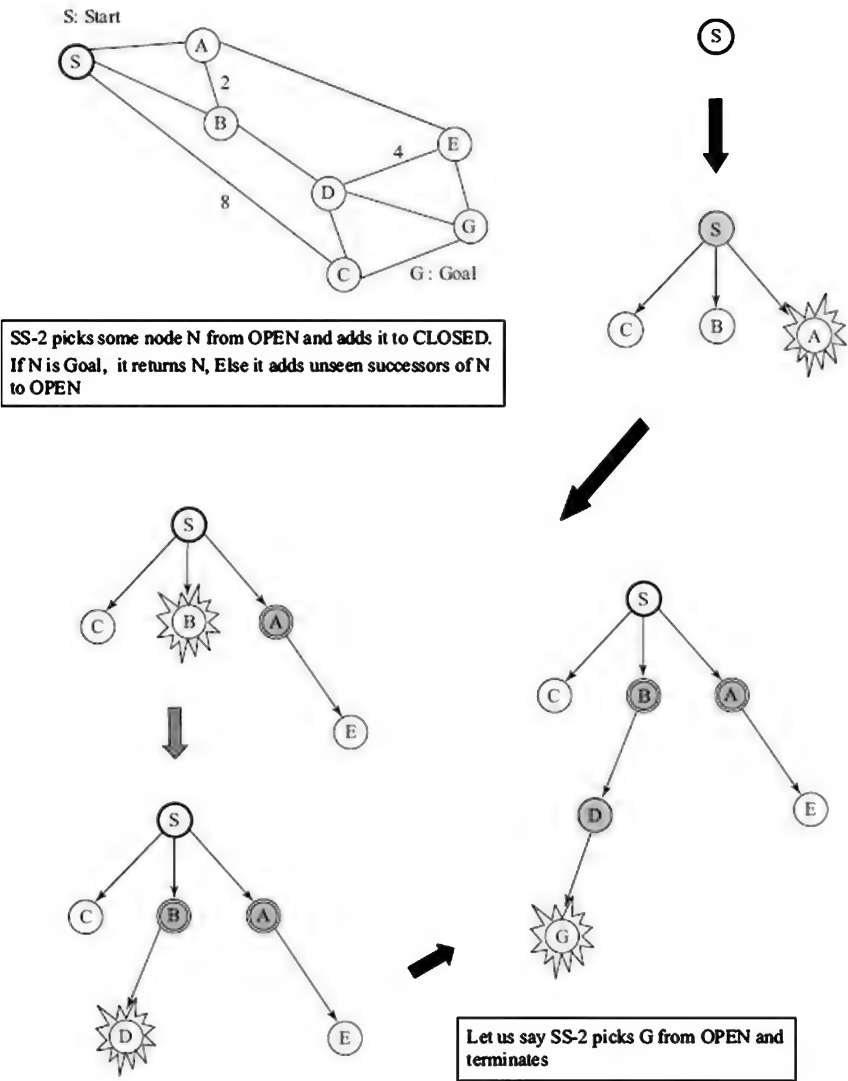


FIGURE 2.14 SS-2 visits each node only once and finds the goal.

The second problem with the above program is that it returns the goal state when it finds it. This is not always problematic though. There are problems in which we are only interested in finding a state satisfying some properties. For example, the *n*-queens problem in which we simply need to show a placement of *n* queens on an $n \times n$ chessboard such that no queen attacks (as defined by rules of the game) any other queen. We can call such problems as *configuration problems*. The other kinds of problems, that we call *planning problems*, are those in which we need a

sequence of moves or the path to the goal state. Clearly, our program is inadequate for the latter kind of problems.

One simple approach is to explicitly keep track of the complete path to each new state. We can do this by modifying the state representation (in the search algorithm only) to store a list of nodes denoting the path. In order to work with the same domain functions *moveGen* and *goalTest*, we need to modify our algorithm. Simple-Search-3 (SS-3) incorporates this change (see Figure 2.15). After picking the node from *OPEN*, it extracts the current state *H*. It tests *H* with *goalTest* and generates its successors using *moveGen*. When the goal node is found, all that the algorithm needs to do is to reverse the node to get the path in the proper order. The function *mapcar* in Figure 2.15 takes a list of lists as an argument and returns a list containing the heads of the input lists. The function *cons* adds a new element to the head of a list.

In some problems, the path information can be represented as part of the state description itself. An example of this is the knight's chessboard-tour problem given in the exercises. One could start by representing the state as a 10×10 array in which the centre 8×8 sub-array, initialized to say 0, would be the chessboard. The starting state would be a square on the chessboard labelled with 1, and subsequent squares could be labelled in the order in which the knight visits them.

```
SimpleSearch3()
1 open ← {(start)}
2 closed ← {}
3 while open is not empty
4   do Pick some node n from open
5     h ← Head(n)
6     if GoalTest(h) = TRUE
7       then return Reverse(n)
8     else closed ← closed ∪ {h}
9         successors ← {MoveGen(h) \ closed}
10        successors ← {successors \ Mapcar(open)}
11        open ← {open \ {n}}
12        for each s in successors
13          do Add Cons(s,n) to open
14 return FAILURE
```

FIGURE 2.15 Algorithm SS-3 stores the path information at every node in the search tree.

The path information stored in the node could also be exploited to check for looping. All that the algorithm would need to do is to check if the new candidate states are not in the path already. While the pruning of nodes will not be as tight as our algorithm above, it would require lesser storage since the set *CLOSED* will no longer be needed. We also remove those successors that are already on *OPEN*. Figure 2.16 shows the *OPEN* and *CLOSED* list for the tiny search problem for the algorithms SS-2 and SS-3.

The search tree as seen by SS-2		SS-3 maintains entire path information at each node on <i>OPEN</i> in the search tree.	
<u>OPEN</u>	<u>CLOSED</u>	The search tree as seen by SS-3	
(S)	()	<u>OPEN</u>	<u>CLOSED</u>
(ABC)	(S)	((S))	()
(BCE)	(SA)	((AS) (BS) (CS))	(S)
(CDE)	(SAB)	((BS) (CS) (EAS))	(AS)
(CGE)	(DABS)	((CS) (DBS) (EAS))	(BAS)
SS-2 terminates when G is picked and found to be the goal.		((CS) (GDBS) (EAS))	(DBAS)
The program returns G.		Again, the search terminates when G is picked.	
		But now the program reverses the path and returns SBDG, which is the path found from S to G.	

FIGURE 2.16 The search trees as seen by SS-2 and SS-3.

In SS-3, *OPEN* contains paths and *CLOSED* contains states. Next, we modify our search function such that both *OPEN* and *CLOSED* store node pairs, representing a node and its parent node in the search tree. Now, all nodes in the search tree have the same structure. We will, however, need to do more work to return the path found. Each node visited has a back pointer to its parent node. The algorithm *reconstructPath* below reconstructs the path by tracing these back pointers until it reaches the start node which has *NIL* as back pointer.

```

ReconstructPath(nodePair, closed)
1 path ← List(Head(nodePair))
2 parent ← Second(nodePair)
3 while parent is not NIL
4     dopath ← Cons(parent, path)
5     nodePair ← FindLink(parent, closed)
6     parent ← Second(nodePair)
7 return path

FindLink(child, closed)
1 if child = Head(Head(closed))
2     then return Head(closed)
3     else return FindLink(child, Tail(closed))

```

FIGURE 2.17 Reconstructing the path from the list of back pointers of each node involves retrieving the parent of the current node all the way back to start whose parent, by definition, is *NIL*. *nodePair* is a pair that contains the goal *g* and its parent node. The functions *List*, *Head*, *Second*, *Tail* and *Cons* for the list data structure.

In the algorithms in Figures 2.17, 2.18 and 2.20, we also move from the *set* representation to a *list* representation. It calls a function *removeSeen* to prune the list of successors, and *makeNodes* to prepare the successors in the form for adding them to *OPEN*. We also make it deterministic by picking the new candidate from the head of the list *OPEN*, since this is an easy operation for most data structures.

2.3 Depth First Search (DFS)

The algorithm *DFS* given below (Figure 2.18) treats *OPEN* like a *stack*. It adds the new candidates at the head of the list. The reason it is called *depth first* is that from the search tree, it selects a node from the *OPEN* list that is *deepest* or farthest from the start node. The candidates are inspected in the last-in-first-out order. This way its focus is on the newer arrivals first, and consequently its characteristic behaviour is that it dives headlong into the search space. Figure 2.19 illustrates the search tree as seen by *DFS*, represented by the two lists *OPEN* and *CLOSED*.

```

DepthFirstSearch()
1 open ← ((start NIL))
2 closed ← ()
3 while not Null(open)
4   do nodePair ← Head(open)
5     node ← Head(nodePair)
6     if GoalTest(node) = TRUE
7       then return ReconstructPath(nodePair, closed)
8     else closed ← Cons(nodePair, closed)
9           children ← MoveGen(node)
10          noLoops ← RemoveSeen(children, open, closed)
11          new ← MakePairs(noLoops, node)
12          open ← Append(new, Tail(open))
13 return "No solution found"

RemoveSeen(nodeList, openList, closedList)
1 if Null(nodeList)
2   then return ()
3   else n ← Head(nodeList)
4         if (OccursIn(n, openList) OR OccursIn(n, closedList))
5           then return RemoveSeen(Tail(nodeList), openList, closedList)
6           else return Cons(n, RemoveSeen(Tail(nodeList), openList, closedList))

OccursIn(node, listOfPairs)
1 if Null(listOfPairs)
2   then return FALSE
3   else if n = Head(Head(listOfPairs))
4     then return TRUE
5     else return OccursIn(node, Tail(listOfPairs))

MakePairs(list, parent)
1 if Null(list)
2   then return ()
3   else return Cons(MakeList(Head(list), parent),
                    MakePairs(Tail(list), parent))

```

FIGURE 2.18 DFS treats *OPEN* like a stack adding new nodes at the head of the list. The function *RemoveSeen* removes any nodes that are already on *OPEN* or *CLOSED*. The function *MakePairs* takes the nodes returned by *RemoveSeen* and constructs node pairs with the parent node, which are then pushed onto *OPEN*.

DFS treats *OPEN* like a stack.

The search tree as seen by *DFS*

<u>OPEN</u>	<u>CLOSED</u>
((S, Nil))	()
((A, S)(B, S)(C, S))	((S, Nil))
((E, A)(B, S)(C, S))	((A, S)(S, Nil))
((D, E)(G, E)(B, S)(C, S))	((E, A)(A, S)(S, Nil))
((G, E)(B, S)(C, S))	((D, E)(E, A)(A, S)(S, Nil))

Again, the search terminates when *G* is picked.

The program reconstructs the path and returns *SAEG*, which is the path found from *S* to *G*. The back pointers are:

G → *E*, *E* → *A*, *A* → *S*

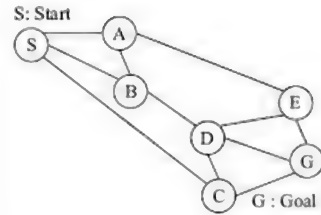


FIGURE 2.19 The search tree as seen by Depth First Search.

2.4 Breadth First Search (BFS)

Breadth First Search (Figure 2.20), on the other hand, is very conservative. It inspects the nodes generated on a first come, first served basis. That is, the nodes form a queue to be inspected. The data structure *Queue* implements the first in, first out order. The *BFS* algorithm is only a small modification of the *DFS* algorithm. Only the order in the append statement where *OPEN* is updated needs to be changed.

```
BreadthFirstSearch()
1 open ← ((start, NIL))
2 closed ← ()
3 while not Null(open)
4   do nodePair ← Head(open)
5   node ← Head(nodePair)
6   if GoalTest(node) = TRUE
7     then return ReconstructPath(nodePair, closed)
8   else closed ← Cons(nodePair, closed)
9   children ← MoveGen(node)
10  noLoops ← RemoveSeen(children, open, closed)
11  new ← MakePairs(noLoops, node)
12  open ← Append(Tail(open), new)
13 return "No solution found"
```

FIGURE 2.20 In *BFS*, the order in append is reversed. *OPEN* becomes a *QUEUE*.

This small difference in the order of adding nodes to *OPEN* produces radically different behaviours from the two search methods. *DFS* dives down expanding the search tree. Traditionally, we draw it going down the leftmost side first. *BFS* on the other hand pushes into the tree, examining it layer by layer. Consider the following small search tree (Figure 2.21) that would be built if every node had three successors, and the total depth was three.

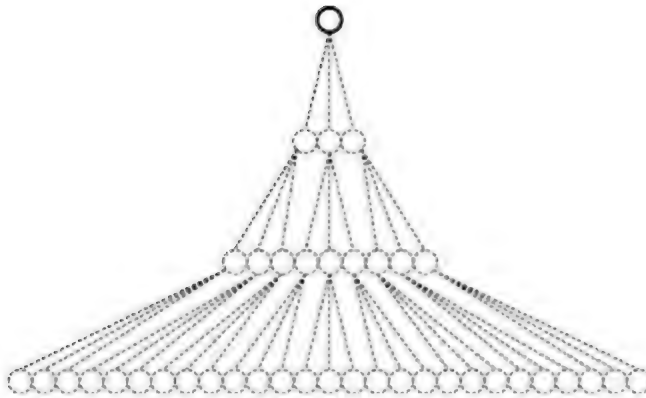


FIGURE 2.21 A tiny search tree.

Let us look at the tree generated by search at the point when the search is about to examine the fifth node, shown in black for both *DFS* and *BFS* in Figure 2.22.

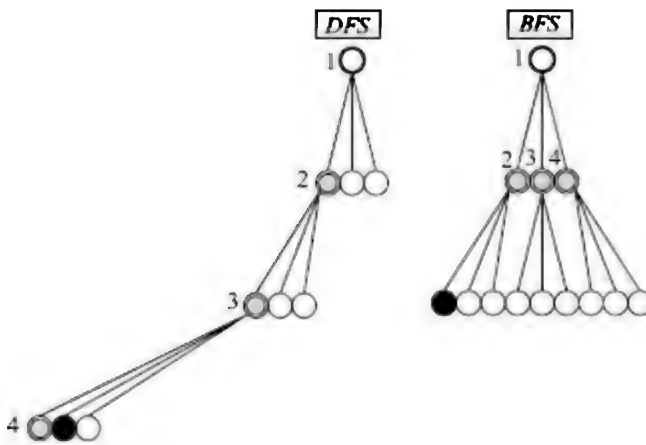


FIGURE 2.22 The search trees generated by *DFS* and *BFS* after four expansions.

DFS dives down the left part of the tree. It will examine the fifth and the sixth nodes and then *backtrack* to the previous level to examine the sibling of the third node it visited. *BFS*, on the other hand, looks at all the nodes at one level before proceeding to the next level. When examining its fifth node, it has started with level two.

How do the two search methods compare? At this point, we introduce the basis for comparison of search methods. The search algorithm is evaluated on the following criteria:

- **Completeness:** Does the algorithm always find a solution when there exists one? We also call a complete search method *systematic*. By this we mean that the algorithm explores the entire search space before reporting failure. If a search is not systematic

then it is *unsystematic*, and therefore incomplete.

- **Time Complexity:** How much time does the algorithm run for before finding a solution? In our case, we will get a measure of time complexity by the number of nodes the algorithm picks up before it finds the solution.
- **Space Complexity:** How much space does the algorithm need? We will use the number of nodes in the *OPEN* list as a measure of space complexity. Of course, our algorithm also needs to store *CLOSED*, and it does grow quite rapidly; but we will ignore that for the time being. There are two reasons for ignoring *CLOSED*. Firstly, it turns out that on the average it is similar for most search methods. Secondly, we also look at search methods which do not maintain the list of seen nodes.
- **Quality of Solution:** If some solutions are better than others, what kind of solution does the algorithm find? A simple criterion for quality might be the length of the path found from the start node to the goal node. We will also look at other criteria in later chapters.

2.5 Comparison of BFS and DFS

How do Depth First Search and Breadth First Search compare? Let us look at each of the criteria described above.

2.5.1 Completeness

Both Depth First Search and Breadth First Search are complete for finite state spaces. Both are systematic. They will explore the entire search space before reporting failure. This is because the termination criterion for both is the same. Either they pick the goal node and report success, or they report failure when *OPEN* becomes empty. The only difference is where the new nodes are placed in the *OPEN* list. Since for every node examined, all unseen successors are put in *OPEN*, both searches will end up looking at all reachable nodes before reporting failure. If the state space is infinite, but with finite branching then depth first search may go down an infinite path and not terminate. Breadth First Search, however, will find a solution if there exists one. If there is no solution, both algorithms will not terminate for infinite state spaces.

2.5.2 Time Complexity

The time complexity is not so straightforward to estimate. If the search space is finite then in the worst case, both the search methods will search all the nodes before reporting failure. When the goal node exists then the time taken to find it depends upon where the goal node is in the state space. Both *DFS* and *BFS* search the space in a predefined manner. The

time taken depends upon where the goal happens to be in their path. Figure 2.23 shows the progress of both the searches on our state space.

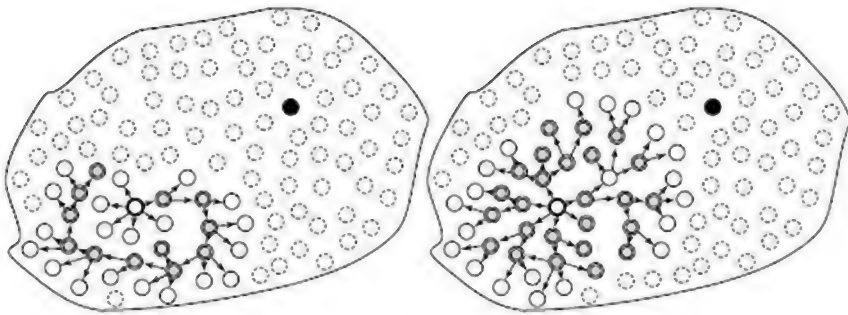


FIGURE 2.23 *DFS* on the left dives down some path. *BFS* on the right pushes slowly into the search space. The nodes in dark grey are in *CLOSED*, and light grey are in *OPEN*. The goal is the node in the centre, but it has no bearing on the search tree generated by *DFS* and *BFS*.

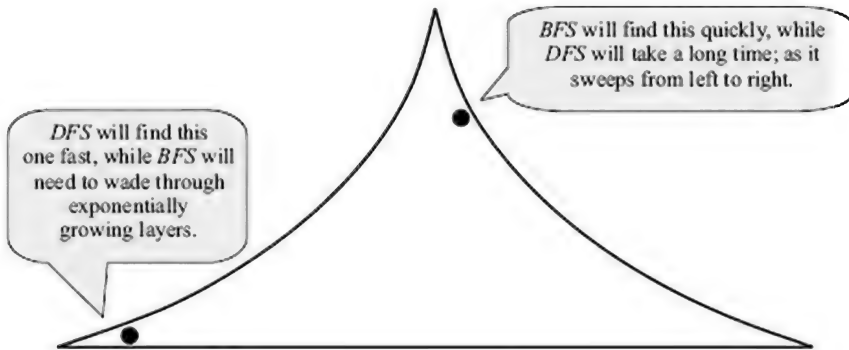


FIGURE 2.24 *DFS* vs. *BFS* on a finite tree.

If the state space is infinite then *DFS* could dive down an infinitely long path and never recover! It would then make no sense to talk of time complexity. Let us assume, for analysis, a domain where the search tree is of bounded depth and of constant branching factor. The tree is shown in Figure 2.24, and depicts two cases that are specifically good for the two searches respectively. A goal node on the bottom left will be found rather quickly by a *DFS* that dives into that area² when it gets going. A goal node in the shallow part of the tree on the right would be found faster by *BFS* that explores the tree level by level. Basically, the *DFS* finds nodes on the left side in the search tree faster, while *BFS* finds nodes closer to the starting node faster.

For quantitative analysis, let us look at a search tree in which the goal is always at the same depth d in a search tree with a fixed branching factor b . Of course, this may not be true in real problems but making this assumption will give us some idea of the complexity of the two algorithms for finding goals at depth d . We further assume that the search tree is of

bounded depth d . An example of such a domain is the n -queens domain, where in each row a queen can be placed in any of the n columns, and a solution will have queens placed in all the n rows. The goal node can be anywhere from the extreme left to the extreme right.

The root node of the search tree is the start node. At level (depth) 1, there are b nodes. At depth 2, there are $b \times b$ nodes, and so on. At level d there are b^d nodes. Also, given a tree of depth d , the number of internal nodes I is given by

$$\begin{aligned}
 I &= 1 + b + b^2 + \dots + b^{d-1} \\
 &= (b^d - 1) / (b - 1)
 \end{aligned}$$

Table (2.1) illustrates the numbers for $b = 10$, and gives us a good idea of how rapidly the search tree grows. At depth = 13, there are 10^{13} leaves and about 1.1×10^{12} internal nodes. An interesting point to note is that the number of internal nodes is significantly smaller than the number of leaves. This means that most of the work by the search is done on the deepest level. This is, of course, a characteristic of an exponentially growing space.

Table 2.1 The number of leaves and internal nodes in a search tree

Depth	Leaves	Internal nodes
0	1	0
1	10	1
2	100	11
3	1000	111
4	10000	1111
5	100000	11111
6	1000000	111111
7	10000000	1111111
8	100000000	11111111
9	1000000000	111111111
10	10000000000	1111111111
11	100000000000	11111111111
12	1000000000000	111111111111
13	10000000000000	1111111111111

Let us now compare the time taken by our two search methods to find a goal node at the depth d . The time is estimated by looking at the size of the *CLOSED* list, which is the number of nodes examined before finding the goal. This assumes that the algorithms take constant time to examine each node. This is strictly not true. Remember that just to check whether a node has been seen earlier, one has to look into the *CLOSED* list. And for that, one has to look into *CLOSED* which grows as more and more nodes are added to it. This is likely to become costlier as the search progresses. Nevertheless, we are only interested in relative estimates, and will adopt the simpler approach of counting the nodes examined before termination. Also, in practice, one might use a *hash table* to store the nodes in *CLOSED*, and then checking for existence could in fact be done in constant time.

DFS

If the goal is on the extreme left then *DFS* finds it after examining d nodes. These are the ancestors of the goal node. If the goal is on the extreme right, it has to examine the entire search tree, which is b^d nodes. Thus, on the average, it will examine N_{DFS} nodes given by,

$$\begin{aligned} N_{DFS} &= [(d+1) + (b^{d-1} - 1)(b-1)] / 2 \\ &= (b^{d-1} - bd + b - d - 2) / 2(b-1) \\ &= b^d / 2 \text{ for large } d \end{aligned}$$

BFS

The search arrives at level d after examining the entire subtree above it. If the goal is on the left, it picks only one node, which is the goal. If the goal is on the right, it picks it up in the end (like the *DFS*), that is, it picks b^d nodes at the level d . On an average, the number of nodes N_{BFS} examined by *BFS* is

$$\begin{aligned} N_{BFS} &= (b^d - 1)(b-1) + (1 - b^d) / 2 \\ &\approx (b^{d-1} + b^d + b - 3) / 2(b-1) \\ &\approx b^d (b-1) / 2b \text{ for large } d \end{aligned}$$

Thus, we can see that the *BFS* does marginally worse than *DFS*. In fact,

$$N_{BFS} / N_{DFS} = (b+1)/b$$

so that for problems with a large branching factor, both tend to do the same work.

Box 2.4: Combex: Unimaginably Large Numbers

Even simple problems can pose a number of choices that we find difficult to comprehend. Consider some search problem where the branching factor is ten, and the solution is twenty moves long. Notice that even the Rubik's cube generates a larger search space. Real world problems will have larger branching factors.

To find the solution, the algorithm may have to examine 10^{20} nodes. How long will this take? Assume that you have a fast machine on which you can check a million nodes a second. Thus, you will need $10^{(20-6)} = 10^{14}$ seconds. Assume conservatively that there are 100000 seconds in a day (to simplify our numbers). Assume a thousand days to a year, and you would need $10^{14-8} = 10^6$ years or a million years.

Surely, you are not willing to wait this long!

In an interesting observation, Douglas Hofstadter (Hofstadter

1986) has talked about our inability to comprehend large numbers. Anything beyond a few thousands is “huge”. Very often two million and two billion look “similar” to us. And numbers like 10^{20} and 10^{30} don’t give us the feel of looking at two numbers, one of which is ten billion times larger than the other. Also, interestingly, George Gamow had written a book called “One, two, three, ... Infinity” (Gamow 71) to highlight a similar observation about comprehending large numbers.

2.5.3 Space Complexity

For assessing space complexity, we analyse the size of the *OPEN* list. That is, the number of candidates that the search keeps pending for examination in the future. Here again, we are ignoring the size of *CLOSED*, which as we know, does grow exponentially, but our interest is in relative measures. Moreover, we will later look at ways to prune the *CLOSED* list in more detail. Figure 2.25 shows the *OPEN* and the *CLOSED* lists at different stages of search in a depth-bounded tree of branching factor 3. The double circle shaded nodes are the ones on *CLOSED*, and the unfilled nodes are the ones on *OPEN*.

DFS

In a general search tree, with a branching factor b , *DFS* dives down some branch, at each level, leaving $(b - 1)$ nodes in *OPEN*. When it enters the depth d , it has at most O_{DFS} nodes in the *OPEN* list, where

$$\begin{aligned} O_{DFS} &= (b - 1)(d - 1) + b \\ &= d(b - 1) + 1 \end{aligned}$$

Thus, the size of *OPEN* is *linear* with depth. In fact, as the search progresses from the left to right, the number of candidates at each level decreases, as shown in Figure 2.26.

Note that the process of backtracking happens automatically when search reaches a dead end. If the examined node has no successors, the next node in *OPEN* is then picked up. This could be a sibling of the node just examined, or if it has no siblings left, a sibling of its parent. Thus, as search exhausts some deeper part of the tree, it automatically reverts to shallower nodes waiting in the *OPEN* list.

BFS

Breadth First Search pushes into the tree level by level. As it enters each level at depth d , it sees all the b^d nodes ahead of it in *OPEN*. By the time it finishes with these nodes, it has generated the entire next level and stored them in *OPEN*. Thus, when it enters the next level at depth $(d + 1)$,

it sees $b^{(d+1)}$ nodes in the *OPEN* list.

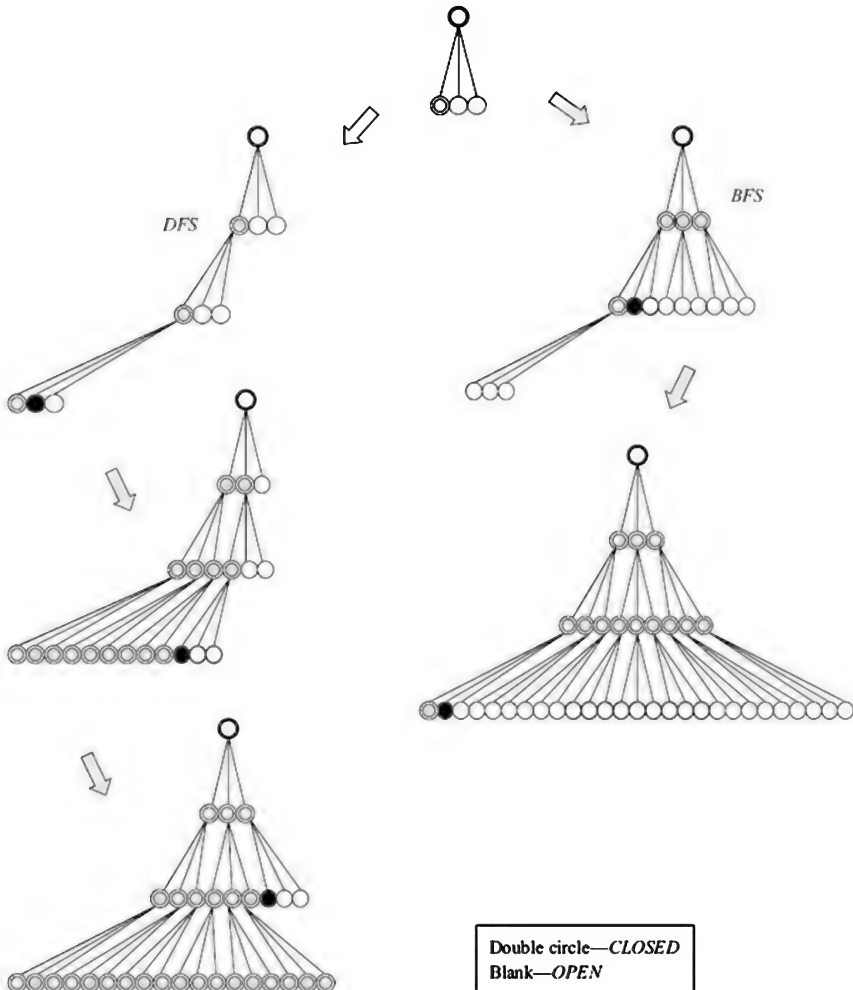


FIGURE 2.25 The *OPEN* and *CLOSED* for *DFS* and *BFS* on the tiny search tree.

This then is the main drawback of *BFS*. The size of *OPEN* grows exponentially with depth. This stands out when one looks at *DFS*, managing its search with an *OPEN* list that grows only linearly with depth.

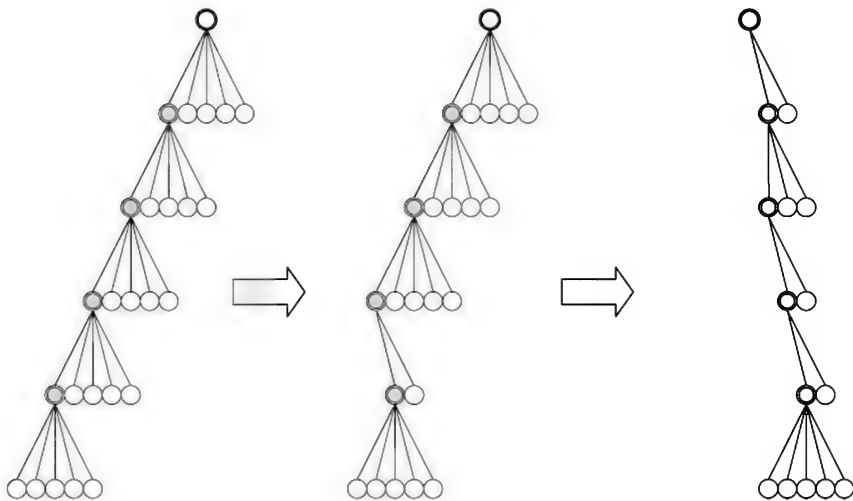


FIGURE 2.26 With a branching factor of 5, at depth 5, there are $5(5 - 1) + 1 = 21$ nodes in the OPEN to begin with. But as DFS progresses and the algorithm backtracks and moves right, the number of nodes on OPEN decreases.

2.6 Quality of Solution

Our formulation of problem solving does not include any explicit cost for the moves or choices made. Consequently, the only quality measure we can talk of is the length of the solution. That is, we consider solutions with smaller number of moves as better.

Where *BFS* loses out on space complexity, it makes up on the quality of the solution. Since it pushes into the search space level by level, the Breadth First Search inspects candidate solutions in increasing order of solution length. Consequently, it will always find the shortest solution. This is a major advantage in many domains.

Depth First Search on the other hand, dives down into the search tree. It backtracks if it reaches a dead end, and tries other parts of the search space. It returns the first solution found, which holds no guarantee that it will be the shortest one. Given a search tree with two goal nodes as shown in Figure 2.24, *DFS* will find the longer solution deeper in the search tree. Thus, it can find a non-optimal solution. On the other hand, since *BFS* pushes into the search tree, it will always find the shortest solution. So, this is one feature where *BFS* is better than *DFS*.

In fact, if the search space is infinite, there is a danger that *DFS* will go into a never-ending path and never return a solution! Consider the task of finding a pair of two integers $\langle m, n \rangle$ which satisfy *some* given property. Let the *moveGen* for each state $\langle x, y \rangle$ return two states $\langle x+1, y \rangle$ and $\langle x, y+1 \rangle$. Let the goal state be $\langle 46, 64 \rangle$ and the start state be $\langle 0, 0 \rangle$. The reader is encouraged to simulate the problem on paper and verify that *DFS* would get lost in an infinite branch.

One way to guard against this possibility is to impose a depth bound on the search algorithm. This means that the search space is prohibited from going beyond a specified depth. This could be done if for some reason one wanted solutions only within a certain length.

2.7 Depth Bounded DFS (DBDFS)

Figure 2.20 shows a depth bounded *DFS* algorithm. It is different from *DFS*, in that it takes a parameter for depth bound. Given any node, it calls *moveGen* only if it is within the given depth bound. The node representation is changed to include depth of node as the third parameter, apart from the parent link, which is the second parameter. The function *makePairs* is modified appropriately to compute the depth of a node, by adding one to the depth of its parent.

The astute reader would have observed that implementing *DBDFS* would have been simpler with the node representation that stored the complete path. One would just have to look at the length of the node to determine whether one is within the allowable bound. And in addition, the task of reconstructing the path is also simplified. Obviously, various design choices will have to be made while implementing solutions for specific problems.

```
DepthBoundedDFS(start, depthBound)
1 open ← ((start, NIL, 0))
2 closed ← ()
3 while not Null(open)
4   do nodePair ← Head(open)
5   node ← Head(nodePair)
6   if GoalTest(node) = TRUE
7     then return ReconstructPath(nodePair, closed)
8   else closed ← Cons(nodePair, closed)
9     if Head(Rest(Rest(nodePair))) < depthBound
10      then children ← MoveGen(node)
11      noLoops ← RemoveSeen(children, open, closed)
12      new ← MakePairs(noLoops, node, Head(Rest(Rest(nodePair))))
13      open ← Append(new, Tail(open))
14 return "No solution found"

MakePairs(list, parent, depth)
1 if Null(list)
2   then return ()
3 else return (Cons(MakeList(Head(list), parent, depth+1)),
4               MakePairs(Tail(list), parent, depth))
```

FIGURE 2.27 Depth Bounded DFS generates new nodes only within a defined boundary.

Performance wise, *DBDFS* is like *DFS* on a finite tree. However, given that the depth bound is artificially imposed, the algorithm is not complete in the general case.

2.8 Depth First Iterative Deepening (DFID)

The algorithm Depth First Iterative Deepening (*DFID*) combines the best features of all the algorithms described above. It does a series of depth first searches with *increasing* depth bounds. Since in every cycle it does a *DFS* with bound incremented by one, whenever it finds a solution it would have found the shortest solution. In this respect, it is like *BFS*. New nodes are explored one level at a time. On the other hand, within each cycle it does a *DBDFS*. Therefore, its memory requirements are those of *DFS*, that is, memory requirements grow linearly with depth.

```

DepthFirstIterativeDeepening(start)
1  depthBound ← 1
2  while TRUE
3    do DepthBoundedDFS(start, depthBound)
4    depthBound ← depthBound + 1

```

FIGURE 2.28 *DFID* does a series of *DBDFS*s with increasing depth bounds.

The high level algorithm described above ignores the possibility of a finite search space with no solution. In that situation, the above algorithm will loop endlessly. The detailed algorithm given below keeps a count of the number of nodes examined in each call of *DBDFS*. If the count is the same in two successive cycles, that is no new nodes are generated, the algorithm *DFID* reports failure.

```

DepthFirstIterativeDeepening(start)
1  depthBound ← 1
2  previousCount ← 0
3  newNodes ← YES
4  repeat
5    count ← 0
6    open ← ((start, NIL, 0))
7    closed ← ()
8    while not Null(open)
9      do nodePair ← Head(open)
10         node ← Head(nodePair)
11         if GoalTest(node) = TRUE
12           then return ReconstructPath(nodePair, closed)
13         else closed ← Cons(nodePair, closed)
14           if Head(Rest(Rest(nodePair))) < depthBound
15             then children ← MoveGen(node)
16                 noLoops ← RemoveSeen(children, open, closed)
17                 new ← MakePairs(noLoops, node,
18                               Head(Rest(Rest(nodePair))))
19                 open ← Append(new, Tail(open))
20                 count ← count + Length(new)
21   if previousCount = count
22     then newNodes ← NO
23   previousCount ← count
24   depthBound ← depthBound + 1
25 until newNodes = NO
26 return "No solution found"

```

FIGURE 2.29 *DFID*—the algorithm in detail.

Thus, the algorithm *DFID* finds the shortest solution using only linear space. Is there a catch somewhere? In a way there is, but only a small one. The *DFID* algorithm does a series of searches. In each search, it explores a new level of nodes. But for inspecting these new nodes, it has to generate the tree all over again. That is, for exploring the new level, it has to pay the additional cost of regenerating the internal nodes of the search tree all over again.

The question one should ask is how significant is the above cost? The new nodes are the leaves of the search tree at the end of that cycle. What then, is the ratio of the number of internal nodes I (the extra cost) to the number of leaves \mathcal{L} (the new nodes) in a tree?

The ratio is the highest for binary trees. The number of internal nodes is just one less than the number of leaves. So, the number of nodes inspected is at most twice as in *BFS*. In general, for inspecting b^d new nodes at level d , one has to inspect $(b^d - 1) / (b - 1)$ extra nodes. The ratio of internal nodes to leaves tends to $1/(b - 1)$ for large d .

Every time *DFID* inspects the leaves at the depth d , it does extra work of inspecting all the internal nodes as well. That is, every time *BFS* scans \mathcal{L} nodes at depth d , *DFID* inspects $\mathcal{L} + I$ nodes. Thus, it does $(\mathcal{L} + I)/\mathcal{L}$ times extra work as compared to *BFS*. This ratio tends to $b/(b - 1)$ for large d , and as the branching factor becomes larger, the number of extra nodes to be inspected becomes less and less significant. A look at Table 2.1 above shows that when $b = 10$, the number of internal nodes is about 11% of the leaves.

Box 2.5: Leaves Vastly Outnumber Internal Nodes

Given a tree with branching factor b , a simple way to compute the ratio of leaves to internal nodes is by thinking of the tree as a tournament in which b players compete, and one winner emerges at each event (internal node). That is, at each (event) internal node, $b - 1$ players are eliminated. Thus, if there are \mathcal{L} players, and only one emerges at the root the number of events (internal nodes), I must satisfy

$$\mathcal{L} = (b - 1) I + 1$$

Thus

$$I = (\mathcal{L} - 1)/(b - 1), \text{ and } I/\mathcal{L} = 1/(b - 1) \text{ for large } \mathcal{L}.$$

Also

$$(\mathcal{L} + I)/\mathcal{L} = b/(b - 1) \text{ for large } \mathcal{L}$$

*The work done by *DFID* can also be computed by summing up the work done by all the Depth First Searches that it is made up of,*

to arrive at the same result.

$$\text{That is } N_{DFID} = \sum_{j=0}^{j+1} d(b^{j+1} - 1)/(b - 1)$$

Thus, the cost of repeating work done at shallow levels is not prohibitive

Thus, the extra work that *DFID* has to do becomes insignificant as the branching factor becomes larger. And the advantage gained over *BFS* is a major one, the space requirement drops from exponential to linear.

The above discussion shows that as search algorithms have to search deeper and deeper then at every level they have to do much more work than at all the previous levels combined. This is the ferocious nature of the monster *CombEx* that problem solving finds itself up against.

Nevertheless, the fact remains that all the three search algorithms explore an exponentially growing number of states, as they go deeper into the search space. This would mean that these approaches will not be suitable as the problem size becomes larger, and the corresponding search space grows. The *DFS* and *DFID* algorithms need only a linear amount of space to store *OPEN*. They still need exponential space to store *CLOSED* whose size is correlated with time complexity. We will need to do better than that.

There are basically two approaches to improve our problem solving algorithms. One is to try and reduce the space in which the algorithm has to search for solutions. This is what is done in *Constraint Propagation*. The other is to guide the search problem solving method to the solution. Knowledge based methods tend to exploit knowledge acquired earlier to solve a current problem. These methods range from fishing out complete solutions from the problem solver's memory, to providing domain insights to a search-based first principles approach.

Observe that the search methods seen so far are completely oblivious of the goal, except for testing for termination. Irrespective of what the goal is or where the goal is in the solution space, each search method described in this chapter explores the space in one single order, which they blindly follow. They can be called *blind* or *uninformed* search methods.

In later chapters, we will find the weapon to fight *CombEx*, our adversary that confounds our algorithms with unimaginably large numbers. Our weapon will be *knowledge*. Knowledge will enable our algorithms to cut through the otherwise intractable search spaces to arrive at solutions in time to be of use to us. In the next chapter, we look at search methods that use some domain knowledge to impart a sense of direction to search, so that it can decide where to look in the solution space in a more informed manner.



Exercises

1. The n -queens problem is to place n queens on an n -by- n chessboard, such that no queen attacks another, as per chess rules. Pose the problem as a search problem.
2. Another interesting problem posed on the chessboard is the knight's tour. Starting at any arbitrary starting point on the board, the task is to move the knight such that it visits every square on the chessboard exactly once. Write an algorithm to accept a starting location on a chessboard and generate a knight's tour.

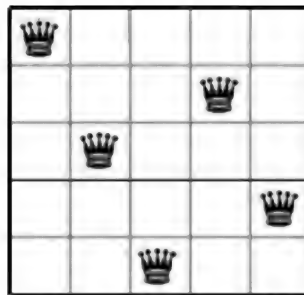


FIGURE 2.30 A 5-queens solution.

Box 2.6: Self Similarity

An interesting observation has been made about chessboards that have sides that are powers of 5. The adjoining figure displays one solution for 6. the 5-queens problem.

If we want to solve the 25 queens problem, then we can think of the 25×25 board as twenty five 5×5 boards. Then, in the five 5×5 boards that correspond to the queens in the above solution, we simply place the 5×5 solution board as shown in Figure 2.31. Thus, the 25×25 board solution is *similar* to the 5×5 board solution. Such similarity to a part of itself is known as *self similarity* and is a key property of fractals (see for example (Barnsley, 1988)). Clouds, coastlines, plants and many artifacts in nature have this property. Observe that with this *knowledge*, arbitrarily large n -queen problems that are powers of five can be solved quite easily, whereas the search would have taken very long.

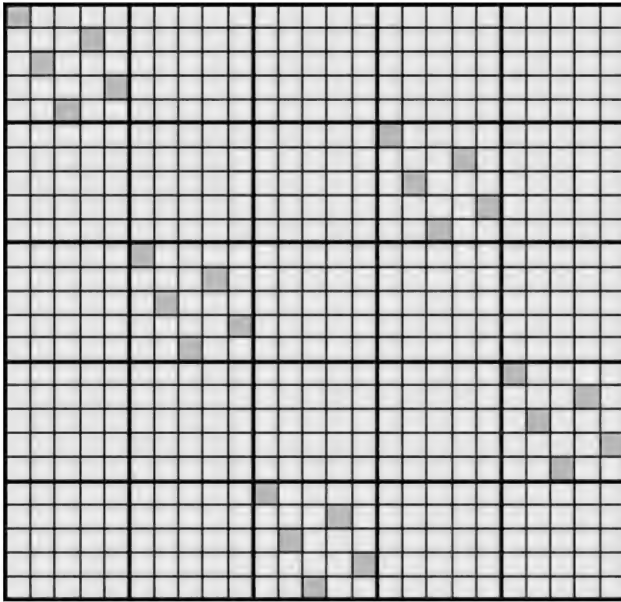


FIGURE 2.31 A 25-queens solution constructed from a 5-queens solution.

3. In the *rabbit leap problem*, three east-bound rabbits stand in a line blocked by three west-bound rabbits. They are crossing a stream with stones placed in the east west direction in a line. There is one empty stone between them.

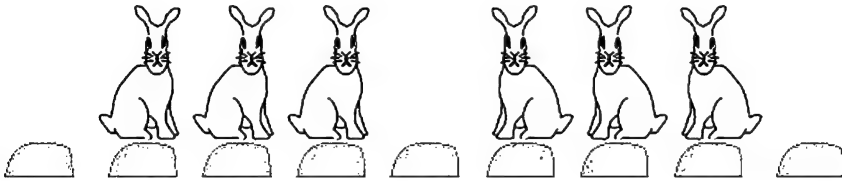


FIGURE 2.32 Rabbits waiting to cross. Each rabbit can jump over one, but not more than that. How can they avoid getting into a deadlock?

The rabbits can only move forward one step or two steps. They can jump over one rabbit if the need arises, but not more than that. Are they smart enough to cross each other without having to step into the water? Draw the state space for solving the problem, and find the solution path in the state space graph.

4. Given that you have a set of shopping tasks to be done at N locations, find a feasible order of the shopping tasks. The constraints are the different closing times of the N shops, given a closing time table. An $N \times N$ array D gives the time it takes to go between shops, where d_{ij} being the time taken from the i^{th} shop to the j^{th} shop.

Assume that the time taken for the actual shopping is negligible.

Hint: Augment the state representation to have time. If the time of reaching a shop is later than its closing time, try another solution.

5. Amogh, Ameya and their grandparents have to cross a bridge over the river within one hour to catch a train. It is raining and they have only one umbrella which can be shared by two people. Assuming that no one wants to get wet, how can they get across in an hour or less? Amogh can cross the bridge in 5 minutes, Ameya in 10, their grandmother in 20, and their grandfather in 25. Design a search algorithm to answer the question.
6. The *AllOut* game is played on a 5 by 5 board. Each square can be in two positions, *ON* or *OFF*. The initial state is some state, where at least one square is *ON*. The moves constitute of clicking on a particular square. The effect of the click is to toggle the positions of its four neighbouring squares. The task is to bring all squares to *OFF* position. Pose the above problem as a state space search problem.
7. Ramesh claims that a given map can be coloured with three colours, such that no adjacent countries have the same colour. The map is represented as a planar graph with nodes as countries and arcs between countries that are adjacent to each other. Design a search program to test his claim for any given problem.
8. In the algorithms given in this chapter, the list *CLOSED* is searched in a linear fashion to find out whether a node has been visited earlier or not. Devise a faster approach to accomplish this task.
9. In the following graph, the node *A* is the start node and nodes *J*, *G* and *R* are goal nodes. The tree is being searched by the *DFID* algorithm, searching left to right. Write the sequence of nodes inspected by *DFID* till termination.

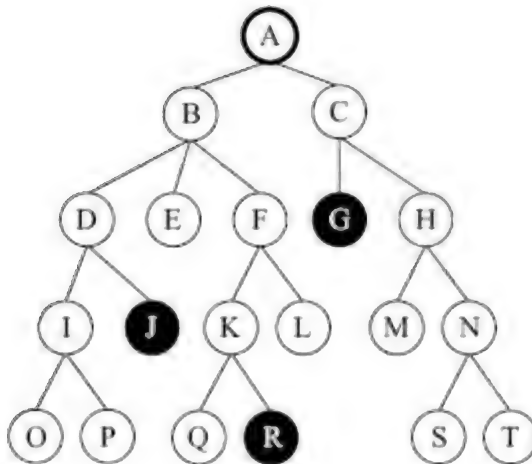


FIGURE 2.33 A small search tree. Assume that the algorithm searches it from left to right.

10. Given the *moveGen* function in the table below, and the

corresponding state space graph, the task is to find a path from the start node *S* to the goal node *J*.

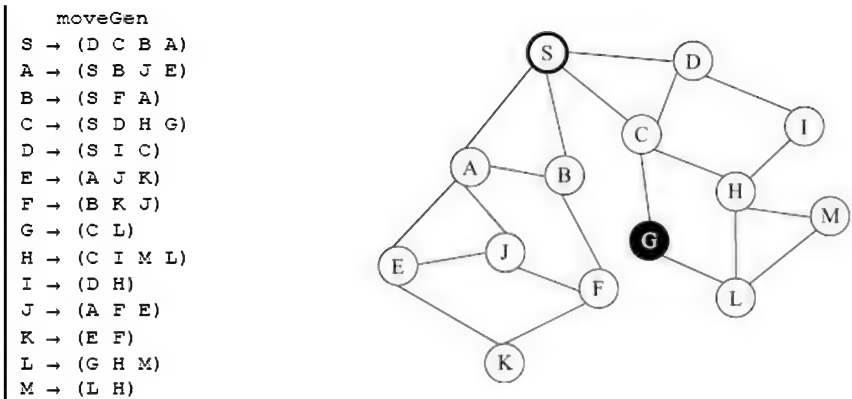


FIGURE 2.34 A small search problem. The task is to find a path from the start node *S* to the goal node *J*.

- Show the *OPEN* and *CLOSED* lists for the *DFS* and the *BFS* algorithms. What is the path found by each of them?
11. The search algorithms depicted in this chapter maintain an *OPEN* list of candidate states. In some domains, the description of a move may need much less space than the description of a state. Show how the *DFS* algorithm can be modified by storing a list of move pairs in the list *OPEN*. Each move pair consists of a forward move and a backward move, to and fro to a successor. What is the space complexity of the resulting algorithm?
 12. How can the same technique be applied to *BFS*? What is the resulting complexity?
 13. Show the order in which *DFID* applies goalTest to the nodes of the above graph. What is the path it finds?
 14. What is the effect of not maintaining the *CLOSED* list in *DFID* search? Discuss with an example. Will the algorithm terminate? Will it find the shortest solution?
 15. Show that the effort expended by *DFID* is the same as its constituent depth first searches.
 16. Which of the following is more amenable to parallelization? *DFS*, *BFS*, or *DFID*? Justify your answer.
 17. Explain with reasons which search algorithms described in this chapter will be used by you for the following problems:
 - (a) A robot finding its way in a maze.
 - (b) Finding a winning move in a chessboard.
 - (c) Finding all winning moves in a chessboard.
 - (d) A sensor trying to route a packet to another sensor (Assume the network topology is known.
- Hint:** Sensors are limited memory devices.) Justify your answers

with a description of your search primitives and state definitions for each problem.

-
- ¹ In an alternate version of the problem, if the missionaries outnumber the cannibals, they will convert them.
 - ² Incidentally, while we have assumed in both *DFS* and *BFS* that the search picks up the leftmost of siblings, this is really determined by the order in which *moveGen* generates them. It is merely easier and more convenient for us to visualize the two algorithms searching from the left to the right.

Heuristic Search

Chapter 3

In the search algorithms described in Chapter 2, the only role that the goal node plays is in testing whether the candidate presented by the search algorithm is a goal or not. Otherwise, the search algorithms are quite oblivious of the goal node. Any intelligent observer watching the progress of the algorithms would soon get exasperated! They *always* go about exploring the state space in the same order, irrespective of the goal to be achieved. They are, therefore, called *blind* or *uninformed*. The *Depth First Search* (see Figure 3.1) dives into the search space, backtracking only if it reaches a dead end. If the search space were infinite, it might just keep going along an endless path. The *Breadth First Search*, on the other hand, ventures out cautiously, going further away, only if it has finished inspecting the nodes the same distance away from the start position. Consequently, it always finds the shortest solution, though its space requirements grow exponentially. *DFID* is basically a sequence of 'depth first searches' masquerading as a breadth first search.

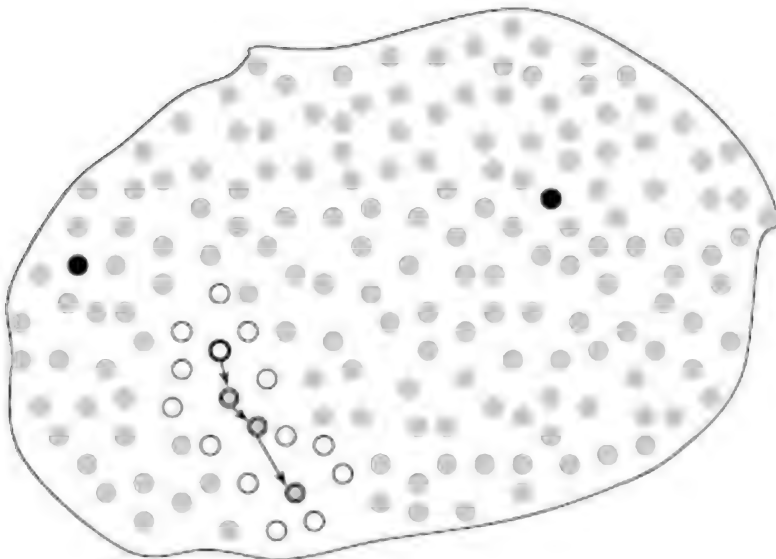


FIGURE 3.1 DFS searches blindly, wherever the goal may be.

The search algorithms described so far maintain a list, called *OPEN*, of candidate nodes. Depending upon whether the algorithm operates *OPEN* as a stack or as a queue, the behaviour is either depth first or breadth first. What we would like is the algorithm to have, instead, some sense of direction. If it could make a guess as to which of the candidates is *more likely* to lead to the goal, it would have a chance of finding the goal node faster. We introduce the notion of a *heuristic function* to enable the search algorithm to make an informed guess.

3.1 Heuristic Functions

As seen in Chapter 2, the time required for search could be exponential in the distance to the goal. The idea of using a heuristic function is to guide the search, so that it *has a tendency* to explore the region leading to the goal. A heuristic function is designed to help the algorithm to pick that candidate from the *OPEN* list that is most likely to be on the path to the goal. A heuristic value is computed for each of the candidates. The heuristic value could be *an estimate* of the distance to the goal from that node, as shown in Figure 3.2. The heuristic function could also embody some knowledge gleaned from human experts that would indicate which of the candidate nodes are more promising. The algorithm then simply has to choose the node with the lowest heuristic value to expand next.

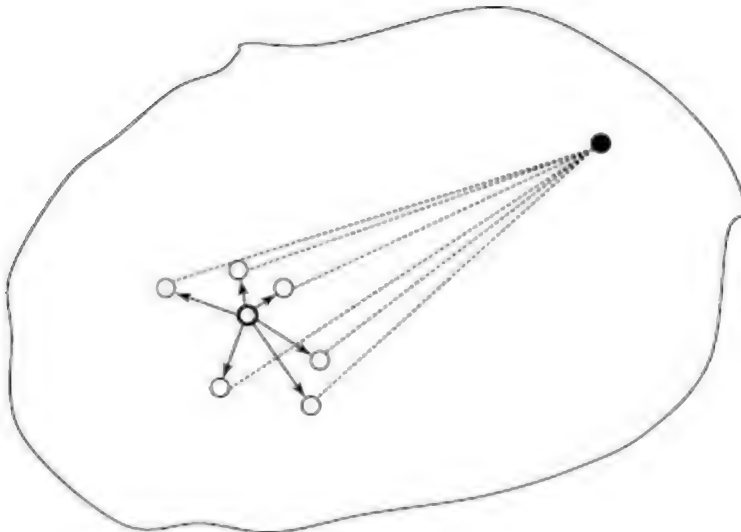


FIGURE 3.2 The heuristic function estimates the distance to the goal.

The word *heuristic* has its roots in the Greek word *εὕρισκω* or *euriskō*, which means “I find, discover”. It has the same root as the word *eureka* from *εὕρηκα* or *heūrēka* meaning “I have found (it)” — an expression, the reader might remember, that is attributed to Archimedes

when he realized that the volume of water displaced in the bath was equal to the volume of his body. He is said to have been so eager to share his discovery, that he leapt out of his bathtub and ran through the streets of Syracuse naked¹.

The heuristic function must not be computationally expensive, because the idea is to cut down on the computational cost of the search algorithm. The simplest way to do this is to make the heuristic function a *static evaluation function* that looks only at the given state and returns a value. It will also have to look at the goal state, or a goal description. Since we expect the move generation function to transform the given state into the goal state via a sequence of moves, the heuristic function has to basically estimate how much of that required transformation still needs to be done for a given state. In other words, it is some kind of a *match* function that computes some *similarity* measure of the current state with the goal state. Such a heuristic function will be domain dependent, and will have to be included into the domain functions, along with the *moveGen* and *goalTest* functions described in Chapter 1. In later chapters, we will also look at the notion of domain independent heuristic functions. These functions estimate the distance to the goal by solving a *relaxed* version of the original problem. The relaxed problems are such that they are simpler to solve, typically being linear or polynomial in complexity. They typically give us a lower bound on the distance to the goal.

Traditionally, the heuristic function is depicted by $h(n)$, where n is the node in question. The fact that the heuristic value is evaluated with respect to a goal state is implicit, and, therefore, the heuristic value should be seen to be for a given problem in which the goal has been specified. To incorporate heuristic values in search, the node-pair representation used will have to be augmented with the heuristic value, so that a node in the search tree will now look like,

searchNode = (currentState, parentState, heuristicValue)

We illustrate the idea of heuristic functions with a few example problems.

In a route finding application in a city, the heuristic function could be some measure of distance between the given state node and the goal state. Let us assume that the location of each node is known in terms of its coordinates. Then a heuristic estimate of distance could be the Euclidean distance of the node from the goal node. That is, it estimates how close to the goal the current state is

Euclidean distance:

$$h(n) = \sqrt{(x_{Goal} - x_n)^2 + (y_{Goal} - y_n)^2}$$

Note that this function gives an optimistic estimate of distance. The

actual distance is likely to be more than the straight line distance. Thus, the Euclidean distance is lower bound on the actual distance. Another distance measure we could use is the *Manhattan distance* or the *city block distance*, which is given below:

Manhattan distance:

$$h(n) = |x_{\text{Goal}} - x_n| + |y_{\text{Goal}} - y_n|$$

This estimates the distance assuming that the edges form a grid, as the streets do in most of Manhattan. Observe that at this point, we are not really interested in knowing the distance accurately; though later we will encounter algorithms that will benefit from such accuracy. At this moment it suffices if the heuristic function can reliably say as to which of the candidates is likely to be *closer* to the goal.

Next, consider the Eight puzzle. The following diagram shows three choices faced by a search algorithm. The choices in the given state are *R* (move a tile right), *U* (up) and *L* (left). Let us call the corresponding states too *R*, *U* and *L*. One simple heuristic function could be simply to count the number of tiles out of place. Let this function be called h_1 . The values for the three choices are:

$h_1(R) =$ (Only 4, 5 and 7 are in place. The rest are in a wrong
6 place.)

$h_1(U) =$ (Again only 4, 5 and 7 are in their final place, but also the
5 blank tile.)

$h_1(L) =$ (2,4,5 and 7 are in place.)
5

Thus, according to h_1 , the best move is either *U* or *L*. Let us look at another heuristic function h_2 that adds up the Manhattan distance of each tile from its destination. The values, counting from the blank tile, and then for tile-1 to tile-8, are:

$$h_2(R) = (2 + 1 + 1 + 3 + 0 + 0 + 2 + 0 + 1) = 10$$

$$h_2(U) = (0 + 1 + 1 + 3 + 0 + 0 + 3 + 0 + 2) = 10$$

$$h_2(L) = (2 + 1 + 0 + 3 + 0 + 0 + 3 + 0 + 1) = 10$$

If one were to think of the heuristic values as obtained from solving a relaxed version of the Eight-puzzle then the first one can be thought of as a problem, where a tile can be moved to its destination in one move, and the second heuristic from a problem where a tile can move even over existing tiles. The values are the total number of moves that need to be made in the relaxed problem(s). Curiously, this more detailed function seems to think that all moves are equally good. We leave it as an exercise for the user to pick the best move in this situation.

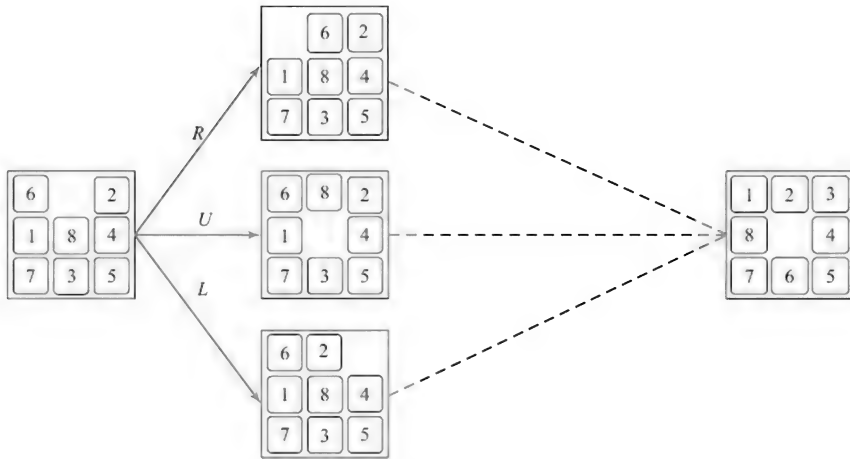


FIGURE 3.3 Which state is closest to the goal?

The Eight-puzzle has its bigger cousins, the 15-puzzle and the 24-puzzle, played on a 4×4 and a 5×5 grid respectively. The state spaces for these puzzles are much larger. The 15-puzzle has about 10000000000000 (or more succinctly 10^{13}) states, while the 24-puzzle has about 10^{25} states. These are not numbers to trifle with. With an algorithm inspecting a billion states a second, we would still need more than a thousand centuries to exhaustively search the state space. Only recently have machines been able to solve the 24-puzzle, and that requires a better heuristic function than the ones we have.

We now look at our first algorithm to exploit the heuristic function, called *Best First Search*, because it picks the node which is best according to the heuristic function.

3.2 Best First Search

We need to make only a small change in our algorithm *Depth First Search* from Chapter 2. We simply maintain an *OPEN* list sorted on the heuristic value, so that the node with the best heuristic value automatically comes to the head of the list. The algorithm then picks the node from the head of *OPEN* as before. Conceptually, this can be written by replacing the line,

$OPEN \leftarrow \text{append}(\text{NEW}, \text{tail}(OPEN))$

with

$OPEN \leftarrow \text{sort}_h(\text{append}(\text{NEW}, \text{tail}(OPEN)))$

In practice, though, it would be more efficient to implement *OPEN* as a priority queue. In addition, one has to make the changes to the search node representation to include the heuristic value, and the consequent

changes in the other functions. These changes are left as an exercise for the reader.

The following figure illustrates the progress of the algorithm on an example problem. The graph in Figure 3.4 represents a city map, where the junctions are located on a two dimensional grid, each being 1 km. A path from the start node *S* to the goal node *G* needs to be found. The search uses the Manhattan distance as the estimate of the distance. The graph depicts the nodes on *CLOSED* with double circles, and the nodes on *OPEN* with single circles, at the instance when the algorithm terminates. Both the sets of nodes are labelled with the heuristic values. The labels near the nodes show the order in which the nodes are expanded. Note that the heuristic function initially takes the search down a path that is not part of the solution. After exploring nodes labelled 2, 3 and 4 in the search order, the search abandons them and goes down a different path. This is characteristic of a heuristic search, and happens because the heuristic function is not “aware” of the entire roadmap. It only has a sense of direction. Perhaps, in this example, there is a river on the way without a bridge at that point. The search then has to explore an alternate route. The back pointers on the edges point to the parent nodes, and the thick edges with back arrows show the path found by the search.

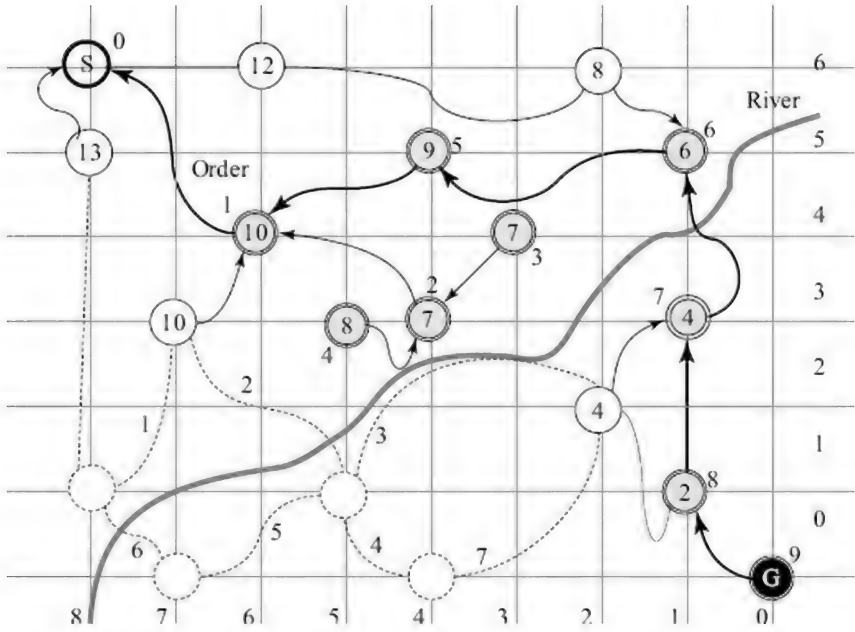


FIGURE 3.4 Best First Searches has a sense of direction, but may hit a dead end too.

What about the performance of the *Best First Search* algorithm? How much does it improve upon the uninformed search methods seen in Chapter 1?

3.2.1 Completeness

Best First Search is obviously complete, at least for finite domains. The reasoning is straightforward. The only change we are making is in the ordering of *OPEN*. It still adds all unseen successors of a node to the list, and like the earlier search algorithms, it will report failure only after *OPEN* becomes empty. That is, only when it has inspected all the candidate nodes. Note that *Best First Search* is systematic too, in the sense that it will inspect all nodes before giving up.

For infinite state spaces, the 'completeness' property will depend upon the quality of the heuristic function. If the function is good enough, the search will still home in on the goal state. If the heuristic function yields no discriminating information (for example if $h(n) = 0$ for all nodes n), the algorithm will behave like its parent algorithm. That is, either *DFS* or *BFS*, depending upon whether it treats *OPEN* like a stack or a queue.

Let us discuss the quality of the solution before looking at complexity.

3.2.2 Quality of Solution

So far we have only talked about the quality of the solution in terms of its length, or the number of moves required to reach the goal. With a simple example shown in Figure 3.5, we can see that it is possible that *Best First Search* can choose a longer solution, in terms of the number of hops. This may happen if the heuristic function measures the difference in terms of some metric which does not relate to the number of steps. Even when we look at other measures for quality, it will be possible to construct examples for which the algorithm picks a sub-optimal solution. This happens mainly because the algorithm only compares two states by estimating the distance to the goal, without taking into account the cost of reaching the two states. Thus, if two states have the same heuristic value, but one of them was more expensive to achieve, the *Best First Search* algorithm has no way of discriminating between the two. As we shall see later (in Chapter 5), incorporating this cost and some conditions on the heuristic will ensure the finding of optimal solutions.

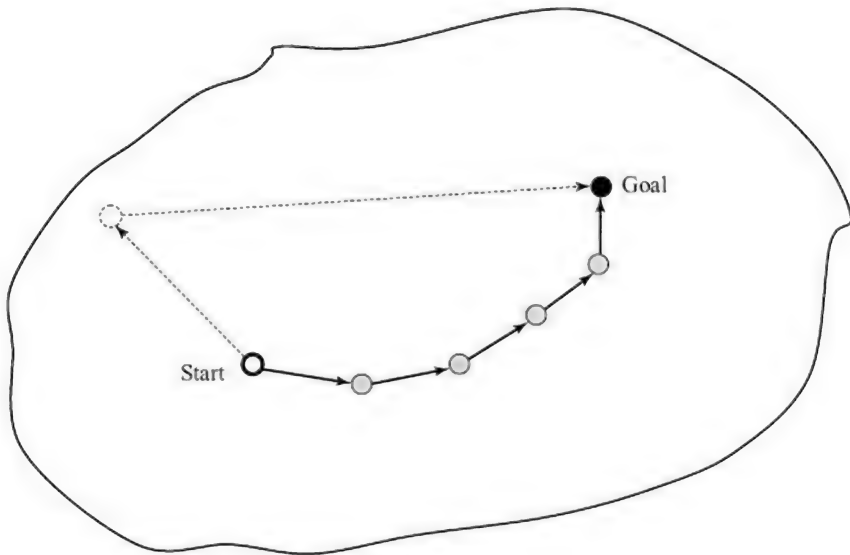


FIGURE 3.5 Best First Search chooses a solution with five moves.

3.2.3 Space Complexity

We have already seen that the search frontier, represented by the list *OPEN*, grows linearly for *DFS* and exponentially for *Breadth First Search*. The search frontier for *Best First Search* depends upon the accuracy of the heuristic function. If the heuristic function is accurate then the search will home in onto the goal directly, and the frontier will only grow linearly. Otherwise, the search algorithm may go down some path, change its mind, and sprout another branch in the search tree (Winston, 1977). Figure 3.6 depicts the frontiers for the three algorithms being discussed. Empirically, it has been found though that for most interesting problems, it is difficult to devise accurate heuristic functions, and consequently the search frontier also grows exponentially in best first searches.

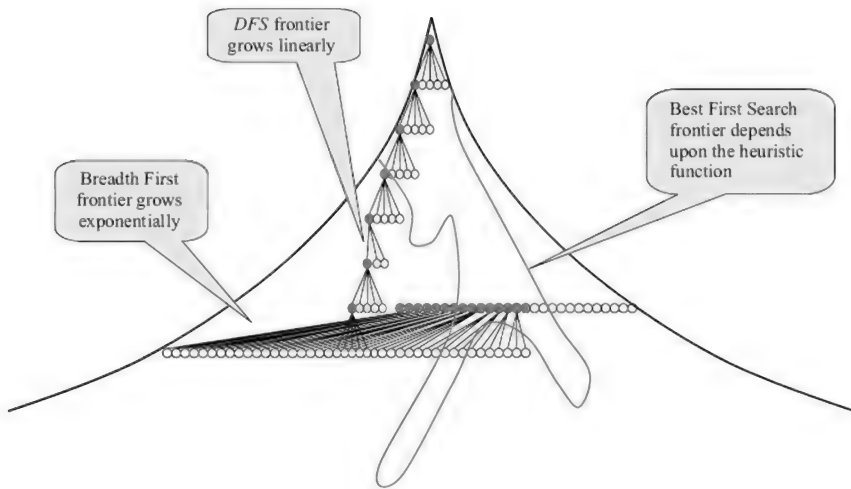


FIGURE 3.6 The Search Frontier is an indication of space requirement.

3.2.4 Time Complexity

Like space complexity, time complexity too is dependent upon the accuracy of the heuristic function. With a perfect heuristic function, the search will find the goal in linear time. Again, since it is very difficult to find very good functions for interesting enough problems, the time required too tends to be exponential in practice.

In both space and time complexity, we have pegged the performance on the accuracy of the heuristic function. In fact, by measuring performance experimentally, we could talk about the accuracy of heuristic functions. A perfect heuristic function would only inspect nodes that lead to the goal. If the function is not perfect then the search would often pick nodes that are not part of the path. We define *effective branching factor* as a measure of how many extra nodes a search algorithm inspects.

$$\text{effective branching factor} = \text{total-nodes-expanded} / \text{nodes-in-the-solution}$$

For a perfect search function, the effective branching factor would tend to be 1. For a very poor heuristic function, one expects the effective branching factor to tend to be much greater than the branching factor of the underlying search space. This is because a really poor function would lead the search away from the goal, and explore the entire space. As an exercise, the reader could implement the two heuristic functions discussed earlier for the Eight-puzzle, and compare the performance over a set of problems. The function that performs better on the average is likely to be better.

A related measure for heuristic functions that has been suggested in literature is called *penetrance* (Nilsson 1998). Penetrance can be defined

as the inverse of effective branching factor. That is:

$$\text{penetrance} = \text{nodes-in-the-solution} / \text{total-nodes-expanded}$$

The best value of penetrance is 1, and if the search does not find a solution, which it might in an infinite search space *with a poor heuristic*, the penetrance value could be said to be zero. Note that both the above measures are for specific search instances. To get a measure of the heuristic function that does not depend upon the given problem, one would have to average the values over many problems.

Best First Search is an *informed* search algorithm. Instead of blindly searching in a predefined manner, it inspects all candidates to determine which of them is most likely to lead to the goal. It does this by employing a heuristic function that looks at the state in the context of the goal, and evaluates it to a number that in some way represents the closeness to the goal. The heuristic functions we have seen are domain dependent. They require the user to define them. Typically, the heuristic function measures closeness by measuring similarity from some perspective. For the route map, finding the similarity is in terms of location; while for the Eight-puzzle, it is in terms of similarity of the patterns formed by the tiles.

In Chapter 10 on advanced methods in planning, we will also see how domain independent heuristics functions can be devised. They will essentially solve simpler versions of the problem in such a way that the time required is significantly smaller. By solving the simpler version for all the candidates, they will be able to give an estimate as to which of the candidates is the closest to the goal. Of course, care has to be taken that the work done by the heuristic function is offset by a reduction in the number of nodes explored, so that the overall performance is better than an uninformed search.

Meanwhile, there exist many interesting real problems that need to be solved, and for which heuristic functions can be devised. For many of these problems, the state representations may be quite complex. There is a need to find algorithms that are easier on space requirements. In the next few sections, we begin by looking at heuristic search algorithms that are guaranteed to have low space requirements. This will enable us to work on complex problems with large search spaces. The cost we may have to pay is in terms of completeness. While *Best First Search* is complete for finite spaces, it becomes impractical if the search spaces are too large. There is still a market for heuristic algorithms to search these spaces, even though they may not *always* find the solution.

3.3 Hill Climbing

If our heuristic function is good then perhaps we can rely on it to drive the search to the goal. We modify the way *OPEN* is updated in *Best First Search*. Instead of,

$OPEN \leftarrow \text{append}(\text{sort}_h(\text{NEW}, \text{tail}(OPEN)))$

we can try using

$OPEN \leftarrow \text{sort}_h(\text{NEW}),$

where, NEW is the list of new successors of the nodes just expanded. Observe that in doing so we have jettisoned the nodes that were added to OPEN earlier. We have burned our bridges as we move forward. A direct consequence of this will be that the new algorithm may not be complete. But having done this, we might as well not maintain an OPEN list, since only the best node from the successors will be visited. The revised algorithm is given in Figure 3.7.

```
HillClimbing()  
1 node ← start  
2 newNode ← Head(Sorth(MoveGen(node)))  
3 while h(newNode) < h(node)  
4   do node ← newNode  
5     newNode ← Head(Sorth(MoveGen(node)))  
6 return newNode
```

FIGURE 3.7 Algorithm Hill Climbing.

Observe that this algorithm has modified our problem solving strategy considerably.

In the search algorithms we saw till now, the termination criterion was finding the goal node. In *Hill Climbing*, the termination criterion is when no successor of the node has a better heuristic value. Searching for the goal has been replaced with optimizing the heuristic value. The problem has been transformed into an optimization problem, in which we seek the node with the lowest heuristic value, rather than one that satisfies the *goalTest* function. This is consistent with the notion that the goal state has the lowest heuristic value, since the distance to the goal is zero. We will also intermittently use the term *objective function* used by the optimization community to refer to the *heuristic function*.

Let us consider the negation of the heuristic function $-h(n)$. Instead of looking for lower values of $h(n)$, we can equivalently say that we are looking for higher values of $-h(n)$. That is, instead of a minimization problem, we have a maximization problem.

As long as the *moveGen* function keeps generating nodes with better (higher) values, the search keeps stepping forward. Of the choices available, the algorithm picks the best one. In other words, the algorithm is performing *steepest gradient ascent*. Imagine you were blindfolded and left on a hillside, and asked to go to the top. What would your strategy be? Presumably, you might test the ground on all sides and take a step in the direction of the steepest gradient. And you would stop when the

terrain in no direction is going upwards. That is precisely what the algorithm *Hill Climbing* is doing; moving along the steepest gradient of a terrain defined by the heuristic function. The idea is illustrated in Figure 3.8 below.

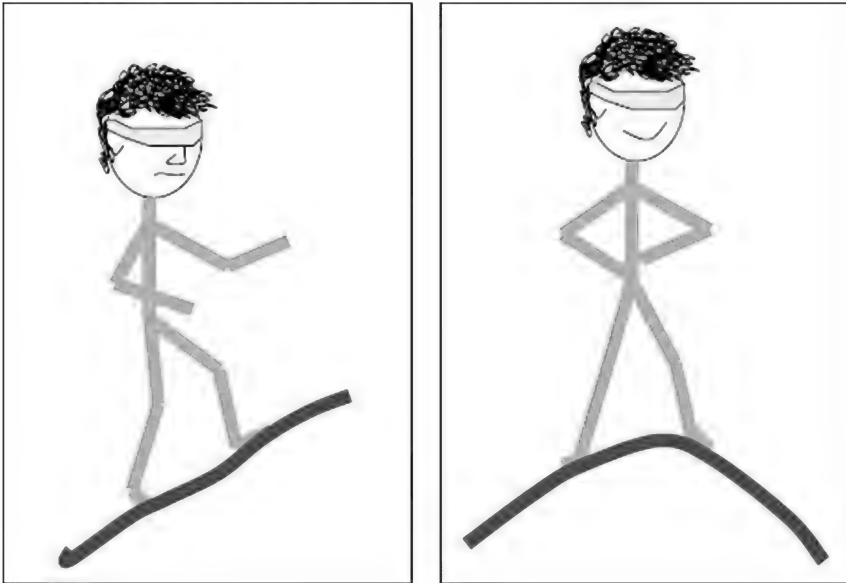


FIGURE 3.8 Hill Climbing.

Observe that while the “terrain” is defined by the heuristic function, and different heuristic functions define different “landscapes”, it is the *moveGen* function that independently determines the neighbourhood available to the search function. The *moveGen* function determines the set of points, both in number and location, in the landscape that become the neighbours of a given node. The heuristic function assigns a heuristic value to each point supplied by the *moveGen* function.

The problem with climbing hills blindfolded is that one does not have a global view. We can only do local moves, and the best we can do is to choose the locally good options. If the hill we are asked to climb had a “smooth” surface then the algorithm will work well. However, mountainous terrain is seldom smooth, and one is more likely than not to end up in the situation depicted in Figure 3.9. Instead of reaching the highest peak, we might end up in a lower peak somewhere on the way. In other words, one has reached a maximum, but only one that is local. If the heuristic function used by *Hill Climbing* is not perfect then the algorithm might well find a local maximum as opposed to a global maximum.

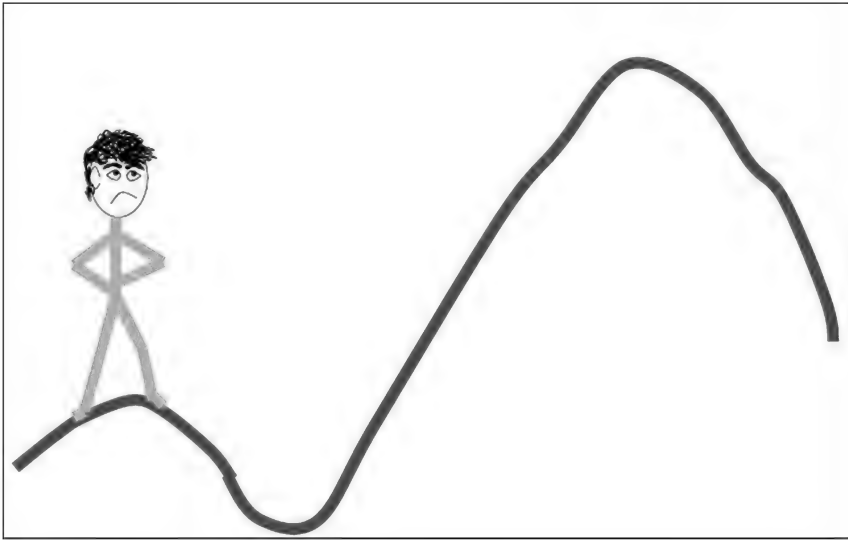


FIGURE 3.9 Stopping at a Local Maximum.

3.4 Local Maxima

We look at an example problem where the choice of a heuristic function determines whether there are local maxima in the state space or not. The blocks world domain consists of a set of blocks on an infinitely large table. There can be only one block on top of each block. The problem is to find a sequence of moves to rearrange a set of blocks, assuming that one can only lift the topmost block from a pile, or keep a block only on the top of a pile. That is, each pile of blocks behaves like a stack data structure. The figure below illustrates a sample problem.

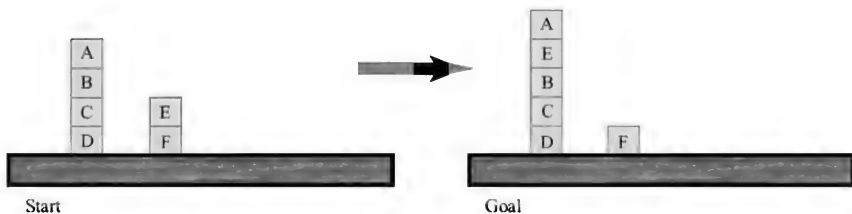


FIGURE 3.10 A blocks world problem.

In the initial state, four moves are possible as shown in Figure 3.11 below. Either block A can be moved, or block E can be moved. The destination is either the other pile or the table. Let us see how two different heuristic functions guide our *Hill Climbing* search algorithm. The two functions (also described in (Rich & Knight, 1990)) differ in the level of detail they look at in a given state.

The first function $h_1(n)$ simply checks whether each block is on the correct block, with respect to the final configuration. We add one for every block that is on the block it is supposed to be on, and subtract one for every one that is on a wrong one. The value of the goal node will be 6. Observe that the heuristic function is not an estimate of the distance to the goal, but a measure of how much of the goal has been achieved. With such a function we are looking for higher values of the heuristic function. That is, the algorithm is performing *steepest gradient ascent*. For the five states, S (start) and its successors P , Q , R , and T , the values are

$$\begin{aligned}h_1(S) &= (-1) + 1 + 1 + 1 + (-1) + 1 = 2 \\h_1(P) &= (-1) + 1 + 1 + 1 + (-1) + 1 = 2 \\h_1(Q) &= 1 + 1 + 1 + 1 + (-1) + 1 = 4 \\h_1(R) &= (-1) + 1 + 1 + 1 + (-1) + 1 = 2 \\h_1(T) &= (-1) + 1 + 1 + 1 + (-1) + 1 = 2\end{aligned}$$

where,

$$h_1(n) = val_A + val_B + val_C + val_D + val_E + val_F$$

Clearly h_1 thinks that moving to state Q is the best idea, because in that state, block A is on block E .

The second heuristic function looks at the entire pile that the block is resting on. If the configuration of the pile is correct, with respect to the goal, it adds one for every block in the pile, or else it subtracts one for every block in that pile. The values for the six nodes we have seen are

$$\begin{aligned}h_2(S) &= (-3) + 2 + 1 + 0 + (-1) + 0 = -1 \\h_2(G) &= 4 + 2 + 1 + 0 + 3 + 0 = 10 \\h_2(P) &= 0 + 2 + 1 + 0 + (-1) + 0 = 2 \\h_2(Q) &= (-2) + 2 + 1 + 0 + (-1) + 0 = 0 \quad h_2(R) = (-3) + 2 + 1 + 0 + (-4) + 0 = -4 \\h_2(T) &= (-3) + 2 + 1 + 0 + 0 + 0 = 0\end{aligned}$$

The first thing to notice is that the heuristic function $h_2(n)$ is much more discriminating. Its evaluation for almost all nodes is different. The second is that its choice of the move is different. It thinks that moving to state (node) P is the best. It is not tempted into putting block A onto block E . The reader will agree that the second function's choice is better.

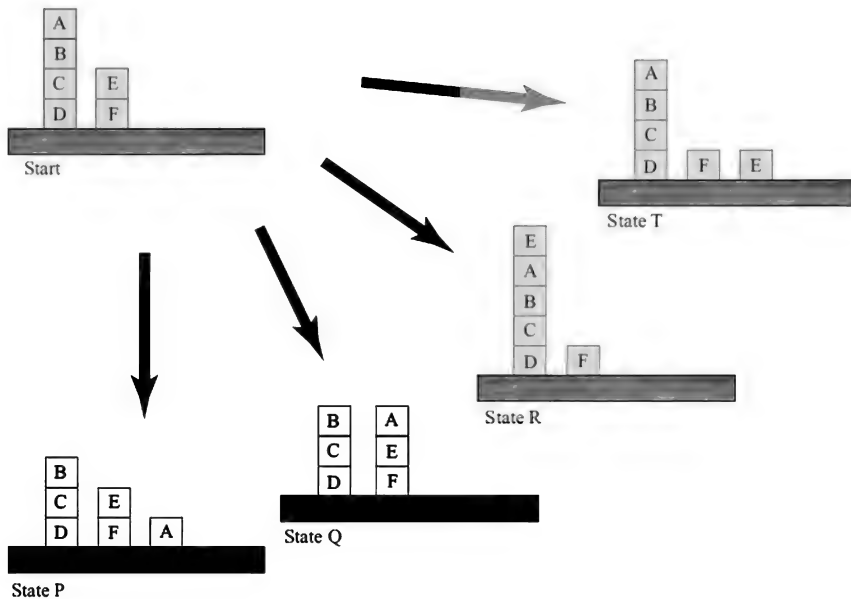


FIGURE 3.11 The first moves possible. A move consists of moving one topmost block to another place.

In both cases, since there is a successor with a better heuristic value, the *Hill Climbing* algorithm will make a move. Let us look at the second move. In the first case, for h_1 , the algorithm is faced with the following choices for the second move, shown in Figure 3.12. We have *named* the states *P* and *Start* the same, though the search will see them as new choices.

The heuristic values of the possible moves are given below. Search is at node *Q*, and all choices have a lower heuristic value. Thus, *Q* is a local maximum, and *Hill Climbing* terminates.

$$h_1(Q) = 1 + 1 + 1 + 1 + (-1) + 1 = 4$$

$$h_1(P) = (-1) + 1 + 1 + 1 + (-1) + 1 = 2$$

$$h_1(S) = (-1) + 1 + 1 + 1 + (-1) + 1 = 2$$

$$h_1(U) = 1 + (-1) + 1 + 1 + (-1) + 1 = 2$$

$$h_1(V) = 1 + (-1) + 1 + 1 + (-1) + 1 = 2$$

Meanwhile, the choices faced by the search using the second heuristic function h_2 , from state *P*, are shown below in Figure 3.13. Like in the previous case, two of the four choices result in states seen earlier.

The heuristic values are

$$h_2(P) = 0 + 2 + 1 + 0 + (-1) + 0 = 2$$

$$h_2(S) = (-3) + 2 + 1 + 0 + (-1) + 0 = -1$$

$$h_2(Q) = (-2) + 2 + 1 + 0 + (-1) + 0 = 0$$

$$h_2(W) = 0 + 2 + 1 + 0 + (-1) + 0 = 2$$

$$h_2(X) = 0 + 2 + 1 + 0 + 3 + 0 = 6$$

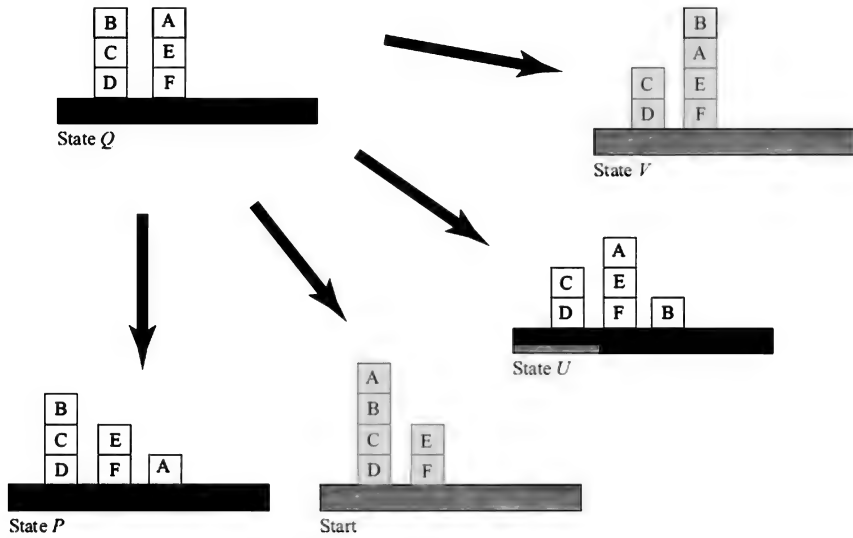


FIGURE 3.12 The choices from state Q.

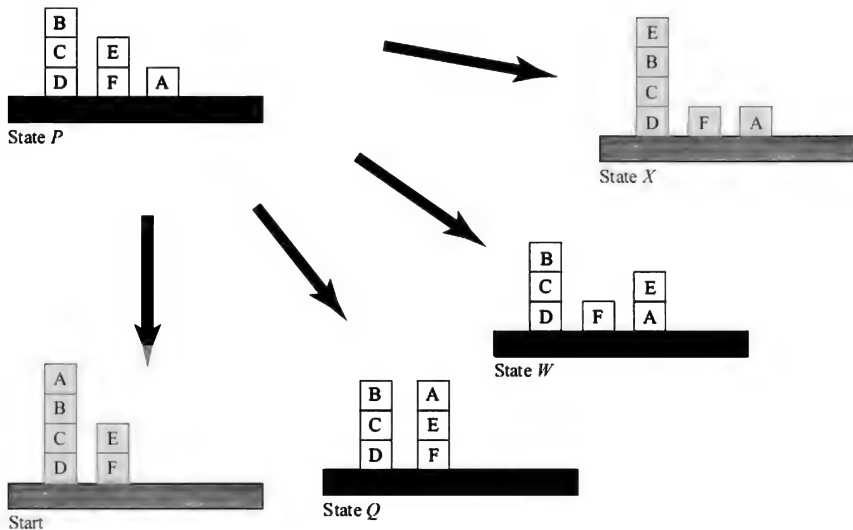


FIGURE 3.13 The choices from state P.

The search has a better option available in state X and moves to it. The reader can verify that in the next move it will reach the goal state.

Thus, we see that the performance of the *Hill Climbing* algorithm

depends upon the heuristic function chosen. One can think of the heuristic function defining a terrain over the search space, with each state having a heuristic value. While the strategy remains the same, that is the *steepest gradient ascent* (or descent, if the heuristic function is such that lower values are better), the performance depends upon the nature of the terrain being defined, as illustrated in Figure 3.14. If the heuristic function defines a smooth terrain, the search will proceed unhindered. On the other hand, if the terrain is undulating then the search could get stuck on a local optimum.

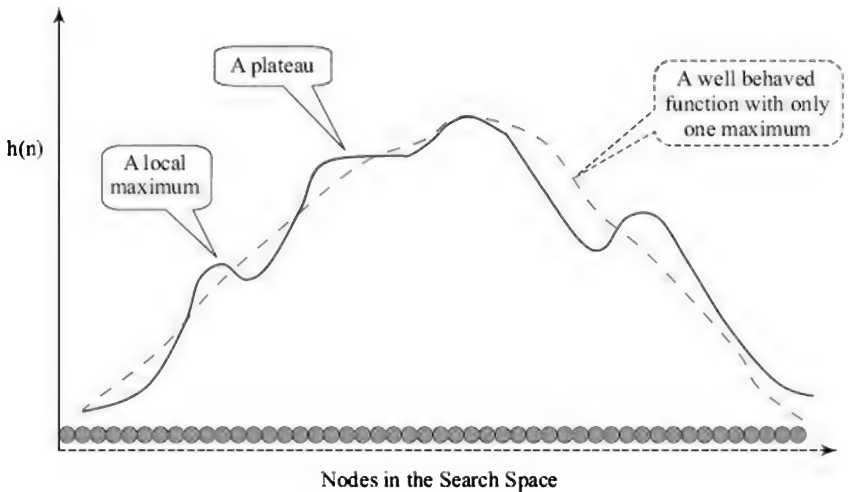


FIGURE 3.14 A well behaved function monotonically improves towards one maximum.

Let us evaluate the *Hill Climbing* algorithm on the four criteria we have been using.

3.4.1 Completeness

Algorithm *Hill Climbing* is not complete. Whether it will find the goal state or not depends upon the quality of the heuristic function. If the function is good enough, the search will still home in on the goal state. If the heuristic function yields no discriminating information (for example, if $h(n) = 0$ for all nodes n), the algorithm will terminate prematurely when it gets stuck.

3.4.2 Quality of Solution

Like *Best First Search*, no guarantee on the quality of the solution can be given.

3.4.3 Space Complexity

This is the *Hill Climbing* algorithm's strongest feature. It requires a constant amount of space. This is because it only keeps a copy of the current state. In addition, it may need some constant amount of memory for storing the previous state and each candidate successor. But overall, the space requirements are constant.

3.4.4 Time Complexity

The time complexity of *Hill Climbing* will be proportional to the length of the *steepest gradient ascent* route from the *Start* position. In a finite domain, the search will proceed along this path and terminate. Thus, one can say that the complexity of *Hill Climbing* is linear.

Overall, one can observe that the performance of *Hill Climbing* is critically dependent upon the heuristic function. The algorithm is an example of a greedy algorithm that makes locally best choices and halts when no locally better option exists. While this may work for some problem domains, in many domains, finding a well behaved heuristic function is not easy. In particular, this is so when a problem can be seen as decomposable into a set of smaller problems. For example, consider the Eight-puzzle or the Rubik's cube kind of problems. Most human solvers tend to decompose the goals into subgoals. For example, while solving the Rubik's cube, one may first do the top row, then the middle row, and finally the bottom row. Each of these subgoals is itself achieved by further decomposition. It is instructive to take a given solution for the Rubik's cube and plot the values of a heuristic function *along the solution path*. Assuming a Manhattan distance like heuristic function, the plot is likely to look somewhat like the one shown in Figure 3.15, but with more local minima. As each subgoal is achieved, the heuristic value reaches a local minimum. But further progress temporarily disrupts the heuristic value, before showing improvement again. Such problems are called problems with *nonserializable subgoals* (Korf 1985) because the subgoals cannot be independently achieved in any serial order. While solving any subgoal, extra care has to be taken to eventually restore any disrupted subgoals.

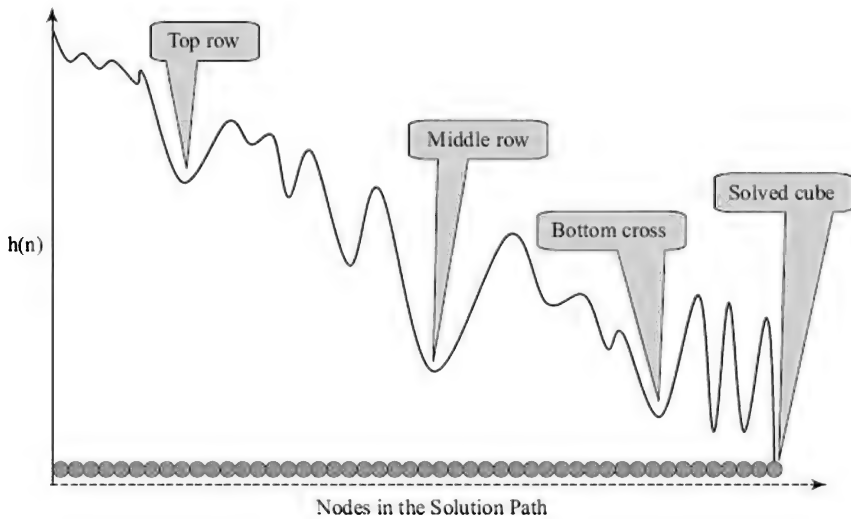


FIGURE 3.15 For human generated solutions of the Rubik's cube, the heuristic value passed through many local minima.

Given that there are going to be problems where the heuristic function will not be well behaved over the domain, there is a need to look for alternate methods to solve such problems. The simplest of them is to increase the memory available by a constant amount, so that more than one option can be kept open. This method is known as the *Beam Search*, described in a later section. But before that, we review the transformation our original problem statement has undergone, and look at an alternate search space formulation.

3.5 Solution Space Search

The problems we have seen so far have been formulated as *constructive* search problems. In a constructive search, we incrementally build the solution. A move consists of extending a given partial solution, and the search terminates when the goal state is found. For example, in the *n-queens* problem, one could start with an empty board, and place one queen at a time. Constructive searches can be both global as well as local. The term *global* and *local* refer to the regions of the search space accessed by the algorithm. The algorithm *Best First Search* is global, in the sense that it keeps the entire search space in its scope. The *Hill Climbing* algorithm, on the other hand, is local, because it is confined only to extending one given path.

Another way to formulate a search problem is with *perturbation* search. In perturbation search, each node in the search space is a *candidate solution*. A move involves perturbation of a candidate solution to produce a new candidate solution. For example, in the *n-queens* problem, the search might start with a random placement of queens, and

then each move may change the position of one or more than one queen. Perturbation searches may be local or global, depending upon whether the algorithm explores the entire search space or only a part of it. Most implementations work with searches looking for local improvements. *Hill Climbing*, the simplest of perturbation search algorithms, does precisely this, and stops when it cannot find a better candidate. Local perturbation search methods are also known as *neighbourhood* search algorithms, because they only search in the neighbourhood of the current node.

Hill Climbing has a termination criterion in which the algorithm terminates when a better neighbour cannot be found. We have already observed that in doing so, it has converted a searching for goal problem into optimization of the heuristic function problem. The *optimization community* refers to the function being optimized as the *objective function*. While focusing on the optimization problem, we will adopt this term.

Box 3.1: The SAT Problem

Consider a Boolean formula made up of a set of propositional variables $V = \{a, b, c, d, e, \dots\}$ (see Chapter 12, propositional logic). For example,

$$F = ((a \vee \neg e) \wedge (e \vee \neg c)) \supset (\neg c \vee \neg d)$$

Each of the propositional variables can take up one of two values: *true* or *false*, known as truth values, and also referred to by 1 and 0, or *T* and *F*. Given an assignment *true* or *false* to each variable in the formula, the formula F acquires a truth value that is dictated by the structure of the formula and the logic connectives used. The problem of satisfiability, referred to as the SAT problem, is to determine whether there exists an assignment of truth values to the constituent variables that make the formula *true*. The formula F given above can be made true by the assignment $\{a = \text{true}, c = \text{true}, d = \text{false}, e = \text{false}\}$ amongst others.

Very often SAT problems are studied in the *Conjunctive Normal Form* in which the formula is expressed as a conjunction of clauses, where each clause is a disjunction of literals, and each literal is a proposition or its negation. The reader should verify that the CNF version of the above formula has only one clause with 4 literals.

$$F = (\neg a \vee \neg c \vee \neg d \vee \neg e)$$

SAT is one of the earliest problems to be proven NP-complete (Cook, 1971). Solving the SAT problem by brute force can be unviable when the number of variables is large. A formula with 100 variables will have 2^{100} or about 10^{30} candidate assignments. Even if we could inspect a million candidates per second, we would need

3×10^{14} centuries or so. Clearly, that is in the realm of the impossible. Further, it is believed that NP-complete problems do not have algorithms whose worst-case running time is better than exponential in the input size.

One often looks at specialised classes of SAT formulas labelled as k -SAT, in which each clause has k literals. It has been shown that 3-SAT is NP-complete. On the other hand, 2-SAT is solvable in polynomial time. For k -SAT, complexity is measured in terms of the size of the formula, which in turn is at most polynomial in the number of variables.

Consider the SAT problem. It involves finding assignments to the set of variables to satisfy a given formula. For example, the following formula has five variables (a, b, c, d, e) and six clauses.

$$(b \vee \neg c) \wedge (c \vee \neg d) \wedge (\neg b) \wedge (\neg a \vee \neg e) \wedge (e \vee \neg c) \wedge (\neg c \vee \neg d)$$

The candidate solutions can be represented as five-bit strings, one bit for the truth value of each variable. For example, 01010 represents the candidate solution $\{a = 0, b = 1, c = 0, d = 1, e = 0\}$. In *solution space search*, we define moves as making some perturbation in a given candidate. For the SAT problem, the perturbation could mean changing some k bits. For the above example, choosing $k = 1$ will yield five new candidates. They are 11010, 00010, 01110, 01000 and 01011. These five then become the neighbours of 01010 in the search space. If we had chosen $k = 2$ then each candidate would have ten new neighbours (we can choose two bits in 5C_2 ways). For 01010, they are: 10010, 11110, 11000, 11011, 00110, 00000, 00011, 01100, 01110, and 01001.

The above example gives us an interesting insight into designing search spaces. For the same search space, or the set of all possible candidate solutions, different neighbourhood functions can be defined by choosing different operators. This would obviously affect the performance of the search algorithm, because all algorithms consider the set of neighbours to select a move. A *sparse* neighbourhood function would imply fewer choices at each point, while a *dense* function would mean more choices. The more dense the neighbourhood, the more expensive it is to inspect the neighbours of a given node. As an extreme in the SAT problem, one could choose an operator that changes all subsets of bits. This would mean that *all* nodes in the search space would become neighbours of the given node, and the search would then reduce to an inspection of all the candidates. Notice that with this all-subsets exchange, there is no notion of a local optimum. When *all* the candidates are neighbours, the best amongst them is the optimum, and that is the global optimum. Conversely, the more sparse the neighbourhood function, the more likelihood of there being a local optima in the search space. The local optima arise because the node (the local optimum) does

not have a better neighbour. That is, better nodes exist in the search space; but the local optimum is not connected to any of them.

The above realization leads to a simple extension of the *Hill Climbing* algorithm, known as the *Variable Neighbourhood Descent*.

3.6 Variable Neighbourhood Descent

In the previous section, we saw that one can define different neighbourhood functions for a given problem. Neighbourhood functions that are sparse lead to quicker movement during search, because the algorithm has to inspect fewer neighbours. But there is a greater probability of getting stuck on a local optimum. This probability of getting stuck becomes lower as neighbourhood functions become denser; but then search progress also slows down because the algorithm has to inspect more neighbours before each move. *Variable Neighbourhood Descent* (VDN) tries to get the best of both worlds (Hansen and Mladenovic, 2002; Hoos and Stutzle, 2005). It starts searching with a sparse neighbourhood function. When it reaches an optimum, it switches to a denser function. The hope is that most of the movement would be done in the earlier rounds, and that the time performance will be better. Otherwise, it is basically a *Hill Climbing* search. In the algorithm in Figure 3.16, we assume that there exists a sequence of *moveGen* functions ordered on increasing density, and that one can pass these functions as parameters to the *Hill Climbing* procedure.

```
VariableNeighbourhoodDescent()  
1  node ← start  
2  for i ← 1 to n  
3    do moveGen ← MoveGen(i)  
4    node ← HillClimbing(node, moveGen)  
5  return node
```

FIGURE 3.16 Algorithm Variable Neighbourhood Descent. The algorithm assumes that the function *moveGen* can be passed as a parameter. It assumes that there are N *moveGen* functions sorted according to the density of the neighbourhoods produced.

3.7 Beam Search

In many problem domains, fairly good heuristic functions can be devised; but they may not be foolproof. Typically, at various levels a few choices may look almost equal, and the function may not be able to discriminate between them. In such a situation, it may help to keep more than one node in the search tree at each level. The number of nodes kept is known as the beam width b . At each stage of expansion, all b nodes are expanded; and from the successors, the best b are retained. The memory requirement thus increases by a constant amount. The search tree

explored by *Beam Search* of width = 2 is illustrated in Figure 3.17.

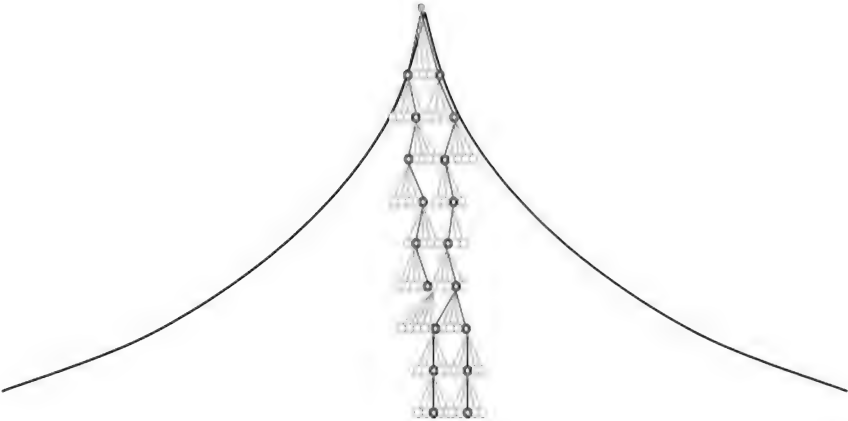


FIGURE 3.17 Beam Search with beam width = 2.

Beam Search is often used in situations where backtracking is not feasible because of other reasons. One of the places where it has been used often is in speech processing. The idea is to combine simpler units of sounds, syllables or phonemes, into bigger units. There are various rules for combining sounds to get meaningful words, and the process has to be done in a continuous (online) mode, producing candidate words as the smaller sound units come in. Since most languages have words that sound similar, where similar symbols that can combine into different word units, a speech processing system benefits by keeping more than one option open. A complete description of various techniques used in speech processing is given in (Huang et al., 2001).

We look at an illustration of *Beam Search* on the instance of SAT discussed earlier in the chapter, reproduced below.

$$(b \vee \neg c) \wedge (c \vee \neg d) \wedge (\neg b) \wedge (\neg a \vee \neg e) \wedge (e \vee \neg c) \wedge (\neg c \vee \neg d)$$

The candidate solutions can be represented as five-bit strings, one bit for the truth value of each variable. Let us choose the starting candidate string as 11111. For the objective function, we choose the number of clauses satisfied by the string. For the instance of the SAT problem given above, this value can range from 0 to 6. Note that this is an example of a maximization problem, because the value of the objective function is maximum for the goal node. We observe that $e(11111) = 3$. Figure 3.18 below shows the progress of *Beam Search* of width 2. For each node in the search space, the candidate string and the heuristic (objective) value are depicted. At each level, the best two nodes are chosen for expansion. Since in our problem the value of the objective function increases by a unit amount, and since the maximum value is 6, the search can move forward only three steps. This is because the *Beam Search*, being an

extension of *Hill Climbing*, is constrained to only move forward to better nodes.

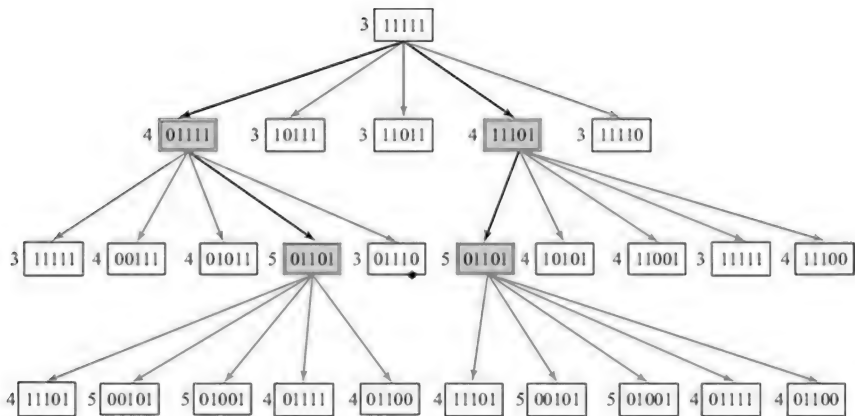


FIGURE 3.18 Beam Search with width 2 fails to solve the SAT problem. Starting with a value 3, the solution should be reached in 3 steps. In fact the node marked * leads to a solution in three steps.

If instead of 11111 we had chosen 01110, the node marked with a star in Figure 3.18, then even the *Beam Search* would have found a path of monotonic increasing values to reach a goal state (there are three different satisfying valuations for the above instance). This means that with a suitable starting point, a solution can be reached. Even with the *Hill Climbing* algorithm, one can reach the solution. An extreme case of this is if one chose a goal as the starting point itself. We explore variations of search algorithms that deploy random choices in the next chapter, and one of them is to randomly choose different starting nodes.

Another way to reach a goal node is to not terminate the search at an optimum point, but to continue looking further. This could be done while keeping track of the best solution found. But if one is to search further beyond an optimum, what should be the terminating criteria? One would then have to devise algorithms that have a criterion decided in advance. Moreover, if one is to get off an (local) optimum, what should be the condition for doing so? Do we simply ignore the idea of gradient ascent? Some possible choices that deal with random moves will be explored in the next chapter. In the following section, we explore a deterministic algorithm that exploits the gradient during search, but is also able to move off optima in search of other solutions.

3.8 Tabu Search

The main idea in *Tabu* search is to augment the *exploitative* strategy of heuristic search with an *explorative* tendency that looks for new areas in the search space (Michalewicz and Fogel, 2004). That is, the search

follows the diktat of the heuristic function as long as better choices are presented. But when there are no better choices, instead of terminating the local search as seen so far, it gives in to its explorative tendency to continue searching. Having got off an optimum, the algorithm should not return to it, because that is what the heuristic function would suggest. *Tabu* search modifies the termination criteria. The algorithm does not terminate on reaching a maximum, but continues searching beyond until some other criterion is met. One way to getting the most out of the search would be to keep track of the best solution found. This would be fairly straightforward while searching the solution space.

Tabu search is basically guided by the heuristic function. As a consequence, even if it were to go beyond a local maximum, the heuristic function would tend to pull it back to the maxima. One way to drive the search away from the maxima is to keep a finite *CLOSED* list in which the most recent nodes are stored. Such a *CLOSED* list could be implemented as a circular queue of k elements, in which only the last k nodes are stored.

In a solution space search where the moves alter components of a solution, one could also keep track of which moves were used in the recent past. That is, the solution component that was perturbed recently cannot be changed. One way to implement this would be to maintain a memory vector M with an entry for each component counting down the waiting period for changing the component. In the SAT problem, each bit is seen as a component, and flipping a bit as a move. Consider a four-variable SAT problem with 5 clauses: $(\neg a \vee \neg b) \wedge (\neg c \vee b) \wedge (c \vee d) \wedge (\neg d \vee b) \wedge (a \vee d)$. Let us say that the period before a bit can be flipped again is 2 time units. This is known as the *Tabu* tenure tt . This means that if one has flipped some bit then it can be flipped back only after two other moves. Assume the evaluation/heuristic function is the number of clauses satisfied. Let the solution vector be in the order $(a \ b \ c \ d)$, and the corresponding memory vector in the same order. Let the starting candidate be $(0 \ 0 \ 0 \ 0)$. The memory vector M is also initialized to $(0 \ 0 \ 0 \ 0)$. This is interpreted that the waiting time for all moves is zero. As soon as a bit is flipped, the corresponding element in M is set to 2, and decremented in each subsequent cycle. At any point, only bits with a zero in the M vector can be considered for a move. The following figure shows the progress of *Tabu* search. After the first expansion, there are two candidates with the same value $e(n) = 4$. The two alternate expansions are shown on the left and right side with different arrows. Note that *Tabu* search would choose randomly between the two. Cells in the top row coloured grey with a thick border show a *tabu* value 2, and grey cells without a thick border depict a *tabu* value 1. These cannot be flipped in that expansion. The *tabu* bits and their values are also shown alongside in the array M . The shaded rows are the candidates that the *Tabu* search moves to.

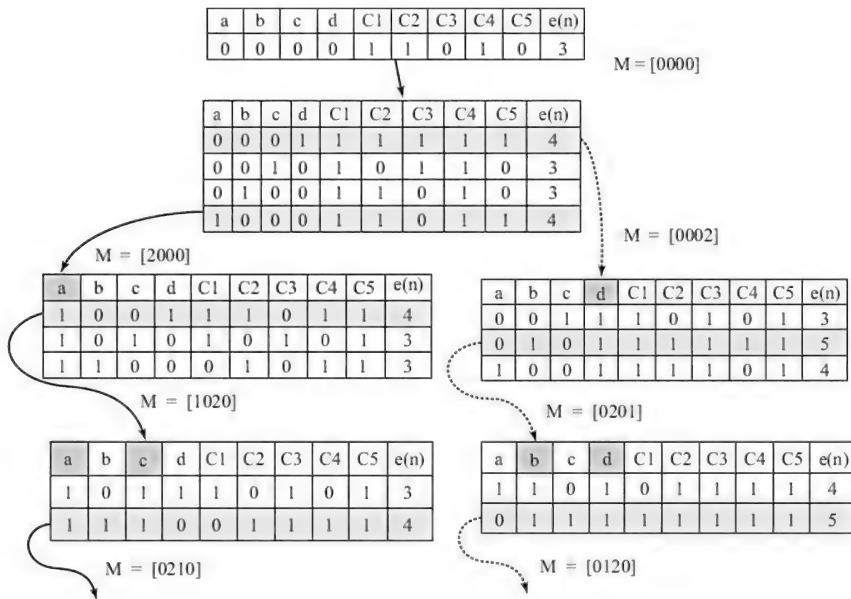


FIGURE 3.19 Two possible paths chosen by the *Tabu* search.

Consider the left branch. Starting with 0000 it goes to 1000. Only the last three bits can be changed and the best choice is 1001. Now in the next move, only the second or third bits can be changed. That is, the state 0001 is also excluded in this path.

The above example illustrates that barring certain moves in a *Tabu* search may also exclude some previously unexplored state. What if one such excluded state is a good one? One could set an aspiration criterion under which moves could overrule the *tabu* placed on them. The criteria could be that all *non-tabu* moves lead to worse nodes, and a *tabu* move yields a value better than all values found so far. Thus, while the *tabu* principle would try and distribute the search amongst different components equitably, the aspiration criteria would still allow potentially best moves to stay in the competition.

Yet another way to diversify a search could be to keep track of the overall frequency of the different moves. Note that this can also be done with a finite memory. Moves that have been less frequently used could be given preference. Imposing a penalty proportional to the frequency on the evaluation value could do this. Thus, nodes generated by frequent moves would get evaluated lower and lower, and the other moves would get a chance to be chosen.

The algorithm *TabuSearch* described below assumes that the candidate solutions have N components, and changing them gives N neighbours, which can be generated by some move generator function called *Change(node, i)* that changes the i^{th} component. The algorithm works with two arrays of N elements. The first called M , keeps track of

the *tabu* list, and is a kind of short term memory. The second called *F*, keeps track of the frequency of changing each component, and serves as a long term memory. The algorithm also assumes the existence of an *Eval(node)* function that evaluates a given node, and the resulting values for the neighbours are stored in an array *Value*. The algorithm written in Figure 3.20 highlights the special aspects of *TabuSearch*, and hence has explicit array computations. The search features are implicit in the calls to the functions *Change* and *Eval*, and the use of the procedure *moveTo(Index)* that finally makes the move and does the bookkeeping is shown in Figure 3.21.

```

TabuSearch(tt)
1 for i ← 1 to n
2   do M[i] ← 0; F[i] ← 0           /* initialize memory */
3 Choose the current node c randomly /* or as given */
4 best ← c
5 while some termination criterion
6   do for i ← 1 to n
7     do /* generate the neighbourhood */
8       tabu[i] ← YES
9       neighbour[i] ← Change(c, i) /*change the i'th component */
10      value[i] ← Eval(neighbour[i])
11      if M[i] = 0
12        then /* if not on tabu list */
13          tabu[i] ← NO
14          bestAllowedValue ← value[i] /*some initial value*/
15          bestAllowedIndex ← i
16 /* BestAllowedValue is best value neighbour that is not on tabu list */
17 /* BestValue is best value amongst all neighbours */
18 bestValue ← value[i]
19 bestIndex ← 1
20 for i ← 1 to n
21   do /* explore the neighbourhood */
22     if value[i] is better than bestValue
23       then bestValue ← value[i]
24          bestIndex ← i
25     if value[i] is better than bestAllowedValue AND tabu[i] = NO
26       then bestAllowedValue ← value[i]
27          bestAllowedIndex ← i
28 if bestAllowedValue is worse than Eval(c)
29 then if bestValue is better than Eval(best)
30   then MoveTo(bestIndex) /*the aspiration criterion */
31 else /* use frequency memory to diversify search */
32   for i ← 1 to n
33     do if tabu[i] = NO
34       then value[i] ← value[i] - Penalty(F[i])
35          /* some initial value */
36          bestAllowedValue ← value[i]
37          bestAllowedIndex ← i
38   for i ← 1 to n
39     do if value[i] is better than bestAllowedValue
40       AND tabu[i] = NO
41       then bestAllowedValue ← value[i]
42          bestAllowedIndex ← i
43   MoveTo(bestAllowedIndex)
44 else /* the best allowed node is an improvement */
45   MoveTo(bestAllowedIndex)

```

FIG 3.20 Algorithm *TabuSearch*.


```

MoveTo(index)
1  c ← neighbour[index]
2  if Value(index) is better than Eval(best)
3    then best ← c
4  F[index] ← F[index] + 1
5  M[index] ← ττ + 1
6  for i ← 1 to n
7    do if M[i] > 0
8      then M[i] ← M[i] - 1

```

FIGURE 3.21 Procedure *MoveTo* makes the move to the new node, and does all the associated bookkeeping operations.

Tabu search is a deterministic approach to moving away from maxima. In the following chapter, we will also explore stochastic search algorithms that have explorative tendencies built into the move-generation process itself. Before doing that, we take a small detour into a knowledge based approach to navigating a difficult heuristic terrain.

3.9 Peak to Peak Methods

The problem solving algorithms seen so far operate at the operator level. Human beings, the best known problem solvers so far, rarely do so. Instead, we often break the problem into sub-problems and solve them. We also remember our problem solving experiences and learn from them. We will look at both these approaches in more detail in later chapters. Here, we look at the idea of macro operators to solve problems in a given domain.

One of the first uses of macro operators was in the *Means Ends Analysis* (MEA) problem solving strategy proposed by Herbert Simon and Alan Newell in their pioneering study on human problemsolving (Newell and Simon, 1972). The MEA strategy operates in a top down manner. Consider the problem of transporting yourself from IIT Madras to the IIT Bombay guesthouse. The *MEA* strategy attempts to identify the largest difference between the current state and the desired state, and looks for a suitable operator to reduce that difference. Say the operator is *flyToMumbai*. The problem solver now has to solve two new problems: One, to reach the Chennai airport, and the other to reach the guesthouse from the Mumbai² airport. In this way, the problem solver works into the details of the solution. We discuss the *MEA* strategy again in Chapter 7.

The idea of macro operators was made more explicit by Richard Korf (1985) and can be illustrated by the way we typically solve the Rubik's cube. Remember that the problem with the Rubik's cube is that it is very difficult to devise a heuristic function that will monotonically drive the search to the solution. Instead, if we were to plot the heuristic function for an expert human solver (see Figure 3.15), we find that there are stages in

the solution where the heuristic values “peaks” and then it gets worse temporarily before attaining a better “peak” (in the figure, the heuristic function goes through better and better local minima). The macro operators proposed by Korf, package the moves between two “peaks” into an operator. The problem solver then has a set of macro operators to work with, and blindly applies each operator without looking at the heuristic value in between. The way I solve the cube, for example, is in the following macro steps:

Get the top cross in place.

Get the top layer in place.

Get the middle layer in place.

Get the bottom corner cubelets in position.

Get the cubelets in correct orientation.

Get the bottom cross in place.

I, myself, fill in the moves for each operator blindly, as taught to me by a friend a long time ago. The operators I learnt were themselves made up of generic operators that could be easily selected. I am essentially using a *knowledge based* approach, not caring to “reinvent the wheel”. The interesting idea that Korf introduced was in devising search algorithms to *learn* the macro operators. Thus, the operators could be learned offline, and at problem solving time, the solution would be found quite quickly.

The idea of macro operators is generalized to the concepts of aggregation and chunking of information, leading to hierarchical problem solving spaces. Human beings work with such packaged information quite naturally, as illustrated by the studies in (Newell and Simon, 1972). In later chapters, we will look at both problem solving and knowledge representation in hierarchical spaces in more detail.

Macro operators illustrate one prominent characteristic of the use of knowledge, which is that at problem solving time, solutions are found very quickly. But acquiring the knowledge that is useful in problem solving is an expensive pre-processing step. Thus, it would make sense to implement knowledge based problem solvers, if the knowledge acquired finds sufficient reuse. In later chapters, we will investigate the means by which problem solvers can benefit from knowledge obtained from human experts. Heuristic functions are a first step in this direction. We will also explore how to build systems that will learn from experience and improve their performance. As of now, most knowledge based systems use representations that are tailored for solving a particular class of problems. This restricts the scope of each system that is built. We will be able to think of generalized, knowledge based programs only after we are able to devise one integrated, representation mechanism common to a variety, if not all applications. These would naturally allow new kinds of algorithms

to exploit a common pool of knowledge. Knowledge representation is thus still an exciting challenge in artificial intelligence. In the meantime, we will continue to build upon a set of general problem solving algorithms that will find a place in the arsenal of any self respecting, intelligent system of the future.

In the next chapter, we look at optimization methods that employ randomness to avoid getting stuck on local maxima.

3.10 Discussion

Heuristic functions give a sense of direction to search algorithms. Given a set of choices that the search process faces, the heuristic function estimates the closeness to the desired goal function. The functions introduced in this chapter are domain dependent heuristic functions. They look at a given state in the context of the goal, and return a value that is an estimate of closeness to the goal. In the *Best First Search*, the *OPEN* list is sorted on the heuristic value, so that the best looking nodes are examined first, yielding a complete algorithm. However, the space requirements are still large. The *Hill Climbing* algorithm burns its bridges as it goes along, discarding the unseen nodes altogether. It has constant space requirements, but is not complete. The *Hill Climbing* algorithm basically exploits the gradient defined by the heuristic function, and may get stuck at nodes that look locally optimum. *Variable Neighbourhood Descent* attempts to increase connectivity in the search space gradually, and to provide more actions as the search moves to better values. *Beam Search* is similar, but always keeps a fixed number of options open. Its space requirements are constant as well, but it has better chances of reaching the goal state. *Tabu* search continues beyond an optimum, keeping a small, constant memory of recent moves. The memory forces the search to explore new areas, often going against the heuristic function. Finally, we observe that knowledge based methods could be used to remember macro operators that could jump from peak to peak on the heuristic terrain. They solve the problem quickly, but the knowledge acquisition process has to be done in advance. All the methods in the chapter are deterministic in nature.



Exercises

1. Devise heuristic functions for the exercises given in Chapter 2.
2. Modify the algorithm for *Depth First Search* (Figure 2.18) to incorporate the heuristic function and do a *Best First Search*.
3. In configuration kind of problems, a goal state is not specified, but a description of the state is given. How would one devise a heuristic function for such problems? Define a heuristic function for the *N*-

- queens problem (A) when one begins with an empty board and places one queen at a time, and (B) when all queens are on the board and the move is a perturbation of a given board position.
4. Write the domain functions for the Eight-puzzle. Devise two heuristic functions h_1 and h_2 . Randomly generate a hundred start states and goal states and run the algorithm with the two heuristic functions. Compute and compare the effective branching factor for the two versions.
 5. It was shown by Johnson and Story (1879) that the Eight-puzzle and Fifteen-puzzle state space is partitioned into two disjoint sets. This means that if the start state is in one partition and the goal state is another, there is no path between the two states. Study the “*parity of permutation*” test described by them, and incorporate it into the Eight-puzzle solver.
 6. Given the two graphs below, show how the algorithms *Hill Climbing* and *Best First Search* explore the graphs. Use Manhattan distance as the heuristic function. Assume that each unit on the grid is 10 kilometres.
 7. For the same task, assuming beam width = 2,
 - (a) Show the order in which the *Beam Search* expands the nodes and the h-value at the time of expansion.
 - (b) Disclose the *OPEN* and *CLOSED* list maintained by the *Beam Search* at each level till the search terminates.

Does the algorithm reach the goal? If yes, what is the path found.
 8. When would you prefer the *Hill Climbing* algorithm to the *Tabu Search* algorithm, and vice versa? Give reasons for your answer.
 9. Given the SAT problem with 5 clauses:
 $(\neg a \vee d) \wedge (c \vee b) \wedge (\neg c \vee d) \wedge (\neg d \vee \neg b) \wedge (a \vee \neg d)$,
 assume that the evaluation/heuristic function is the number of clauses satisfied. Let the solution vector be in the order $(a \ b \ c \ d)$. Let the starting candidate be $(0 \ 0 \ 0 \ 0)$.

Show 3 expansions of the *Tabu* search assuming $tt = 2$ (that is *tabu tenure* is 2). Show the new candidate as well as the *tabu* moves at each of the three stages. Also, draw a table showing which clauses are true for each candidate.
 10. Write a program to randomly generate k -SAT problems. The program must accept values for k , m the number of clauses in the formula, and n the number of variables. Each clause of length k must contain distinct variables or their negation. Instances generated by this algorithm belong to *fixed clause length model* of SAT and are known as *Uniform Random k-SAT* problems.

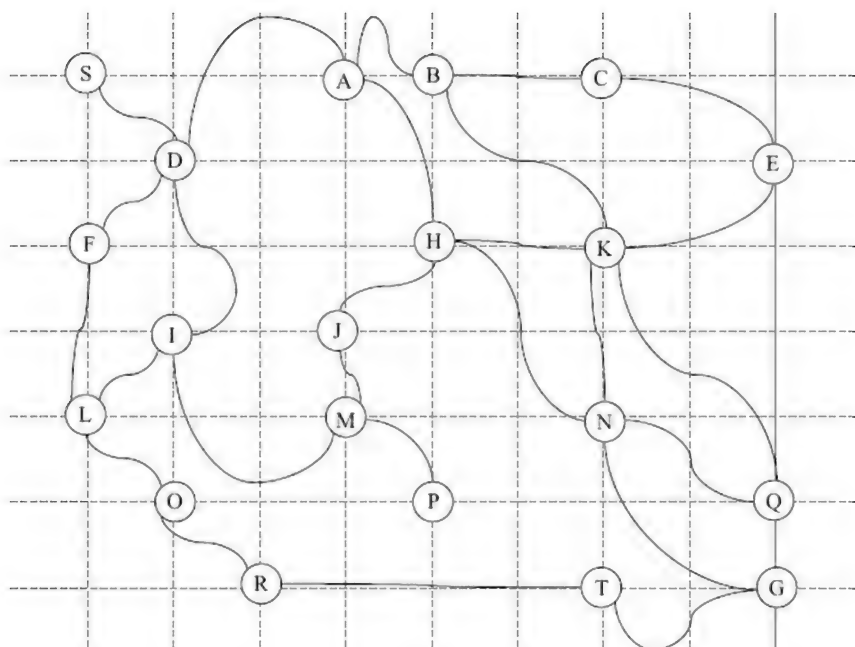


FIGURE 3.22 Problem graph 1. Locations A to Q are located on a grid with each square side being 10 km.

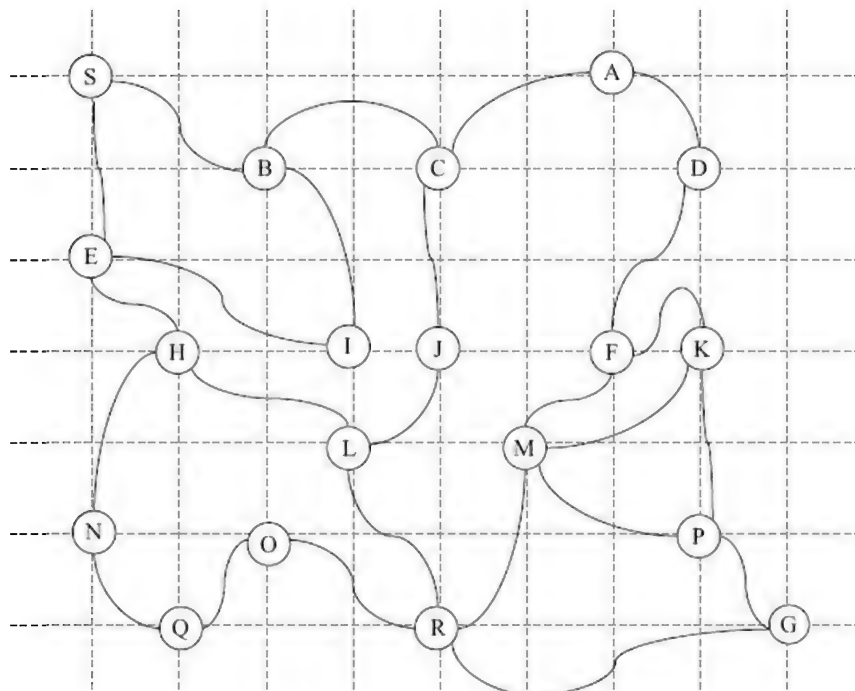


FIGURE 3.23 Problem graph 2. Locations A to Q are located on a grid with each square side being 10 km.

11. Write programs to solve a set of *Uniform Random 3-SAT* problems for different combinations of m and n , and compare their performance. Try the *Hill Climbing* algorithm, *Beam Search* with beam widths of 3 and 4, *Variable Neighbourhood Descent* with 3 neighbourhood functions, and *Tabu Search* with neighbourhood functions changing up to 2 bits at a time. (It has been shown that for a given n when the number of clauses is small, the problems are underconstrained trivially solvable; when m is large they are not likely to have a solution; and around $m = 4.24n$ there is sudden drop in the probability of solving them (Mitchell et al., 1992).)

¹ [http://en.wikipedia.org/wiki/Eureka_\(word\)](http://en.wikipedia.org/wiki/Eureka_(word))

² IIT Madras is in Chennai (formerly Madras) and IIT Bombay, in Mumbai (formerly Bombay).

Randomized Search and Emergent Systems

Chapter 4

Escaping Local Maxima

We have seen that most real world problems create search spaces that are too huge to explore fully. The most studied problems with large solution spaces are *SAT* and *TSP*. A *SAT* problem on a hundred variables has 2^{100} candidate assignments. If we had a machine inspecting a thousand of these per second, and if the machine was started at the time of the Big Bang, fourteen billion years ago when the (current) universe came into being, it would have finished less than one percent of its task by now. Clearly, inspecting all the candidates is not a viable option. Similar search spaces crop up with most problems we formulate. With optimization problems like the *TSP*, the difficulty is compounded by the fact that we would usually¹ not recognize that a solution is optimal, even if we were to find it. Observe that we also see the *SAT* problem as an “optimization” problem during search, in which we attempt to optimize the heuristic value of a candidate solution. Only, in this case, the optimal value is recognized easily; for example when the number of unsatisfied clauses is zero, and we can terminate the search. Complete search not being a viable option for large problems, search methods like *Hill Climbing* and *Tabu Search* work with bounded memories. While *Hill Climbing* is conceptually simple, it can get stuck on a local optimum. In *Tabu Search*, the attempt is to diversify search, as opposed to only following the steepest gradient, often moving to a node that is not the best successor, or even a better one. This allows it to move away from local optima, and the possibility of moving towards the global one is kept open.

The steepest gradient ascent attempts to *exploit* the gradient information (Michalewicz and Fogel, 2004). To this, the *Tabu Search* adds an *explorative* component by trying to push the search into newer areas. It does this in a deterministic way, keeping a *Tabu* list of recent moves to be avoided. In this chapter, we look at randomized approaches to promoting exploration. First, we look at a way to randomize the *Hill Climbing* algorithm. Then, we look at other approaches motivated by the way random moves made in nature can lead to build up and preservation of good solutions.

4.1 Iterated Hill Climbing

Working in the solution space, our search algorithms perturb current

candidate solutions in search of better ones. The notion of a start node and a goal node in the state space is replaced by optimizing the value of the evaluation or heuristic function. The start node in the search tree has no particular significance when we are searching in the solution space. It is simply a candidate solution we begin with. For the *Hill Climbing* algorithm, this is the starting point of the steepest gradient ascent (or descent, if the problem is of minimization). Once the starting point is decided, the algorithm moves up (or down) along the steepest gradient, and terminates when the gradient becomes zero. The end point of the search is determined by the starting point, and the nature of the surface defined by the evaluation function. If the surface is monotonic then the end point is the global optimum; otherwise the search ends up at an optimum that is in some sense closest to the starting point, but may only be a local optimum. *Iterated Hill Climbing* exploits this property by doing a series of searches from a set of randomly selected different starting points. The hope is that the best value found by at least *one* of the searches will be the global optimum. The algorithm can be written as shown in Figure 4.1.

```

IteratedHillClimbing(n)
1  node ← random candidate solution
2  bestNode ← node
3  for i ← 1 to n
4    do node ← random candidate solution
5      newNode ← Head(Sorth(MoveGen(node)))
6      while h(newNode) > h(node)    /* ">" for maximization */
7        do
8          node ← newNode
9          newNode ← Head(Sorth(MoveGen(node)))
10     if h(newNode) > h(bestNode)
11       then bestNode ← newNode
12  return bestNode

```

FIGURE 4.1 Algorithm *Iterated Hill Climbing (IHC)* for a maximization problem. It does a number of *Hill Climbing* runs from random starting points.

The *Iterated Hill Climbing* approach will work well if the surface defined by the evaluation function is smooth at the local level, with perhaps a small number of local optima globally. Such a surface is illustrated in Figure 4.2.

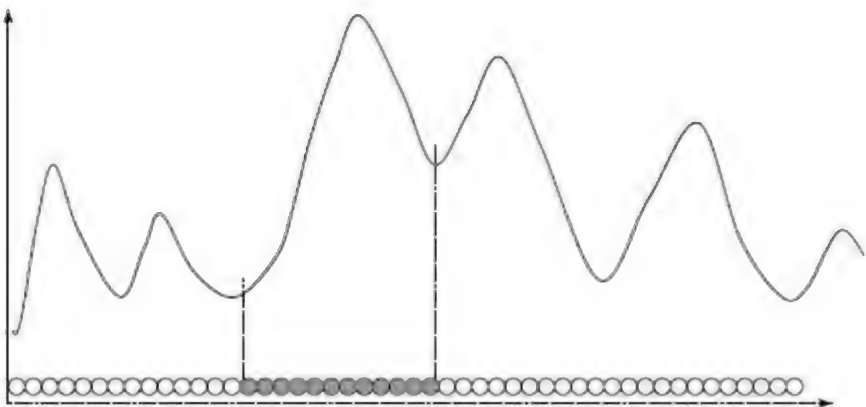


FIGURE 4.2 A smooth surface with a small number of local optima is well suited for *Iterated Hill Climbing*. A random starting point in any iteration from any of the shaded nodes would lead to the global maximum.

The *IHC* algorithm has the same space requirements as *Hill Climbing*. Both need a constant amount of space to run. The difference is that for different runs on the same problem, the *Hill Climbing* algorithm will always return the same result; while the *IHC* algorithm may return different results. This is because its performance is determined by the random choice of the random starting points. Figure 4.2 shows a set of shaded nodes from where the steepest gradient leads to the global maximum. Let us call this set the *footprint* of the *HC* algorithm. If in one of the outer loop iterations of *IHC*, a point in *this* region is chosen as the starting point then the algorithm will find the optimal solution. The likelihood of *IHC* finding the global optimum depends upon the size of the footprint. The larger the footprint is, the greater is the chance of starting in it, and finding the optimum. If the footprint covers the entire search space then *HC* itself will work. As the footprint gets smaller, one would need a larger number of iterations in the outer loop to have a good chance of finding the optimum. If the footprint is very small then the number of iterations in the outer loops may become prohibitively large, and one may have to look for an alternative approach. Such an evaluation function surface is depicted below in Figure 4.5, and the next section describes an approach that might work better there.

4.2 Simulated Annealing

The algorithms we have seen so far have all depicted deterministic search moves. The *IHC* algorithm above introduces a randomized aspect by choosing a series of random starting points followed by deterministic moves in the search space. We now look at the possibility of making random moves. The effect of this would be that even from the same starting point, the search may proceed differently for different runs of the algorithm. A completely random procedure is the *Random Walk*, in which the search process makes random moves in the search space in complete disregard of the gradient. The algorithm is shown below in Figure 4.3.

```
RandomWalk()  
1  node ← random candidate solution or start  
2  bestNode ← node  
3  for i ← 1 to n  
4      do node ← RandomChoose(MoveGen(node))  
5          if h(node) > h(bestNode)  
6              then bestNode ← node  
7  return bestNode
```

FIGURE 4.3 *RandomWalk* explores the search space in a random fashion. Function *RandomChoose* randomly picks one of the successors of the current node. The above algorithm has n random moves.

Given a surface defined by the evaluation function, *Random Walk* and *Hill Climbing* are two extremes of local search. *Random Walk* relies on *exploration* of the search space. Its performance is dependent upon time. The longer you explore the space, the more the chances of finding the global optimum. Because it is totally oblivious of the gradient, it never gets stuck on a local optimum. *Hill Climbing*, on the other hand, relies on the *exploitation* of the gradient. Its performance depends upon the nature of the surface. It terminates on reaching an optimum. If the surface has a monotonic gradient towards the global optimum, *HC* will quickly reach that. Otherwise, it will reach the nearest local optimum.

Simulated Annealing (SA) is an algorithm that combines the two tendencies, *explorative* and *exploitative*, of the two search methods. The basic idea is that the algorithm makes a probabilistic move in a random direction. The probability of making the move is proportional to the *gain* of value made by the move. Traditionally, this gain is associated with energy² and we use the term ΔE to represent the change in the evaluation value. The larger the gain, the larger is the probability of making the move. The algorithm will make a move even for negative gain moves with nonzero probability, though this probability decreases as the move becomes worse. The point is that the algorithm will make moves *against* the gradient too, but will have a higher probability of making better moves. Consider a maximization problem from which three states *A*, *B* and *C* are shown in Figure 4.4. Both *A* and *B* are maxima with *B* having a higher evaluation value. If the algorithm has to move from local maximum *A* to *C*, it will have a negative gain of ΔE_{AC} . Likewise, if it has to move from *B* to *C*, it will have to go through a negative gain of ΔE_{BC} . Since this is larger than ΔE_{AC} , it is likely that the algorithm moves from *A* to *C* more often than from *B* to *C*. Again, since the positive gain from *C* to *B* is higher; the algorithm is more likely to move from *C* to *B* than to *A*. That is, it is more likely that the algorithm will move from *A* to *B* than in the other direction. Then, given a series of local maxima of increasing magnitude, the search is likely to move towards the better ones over a period of time.

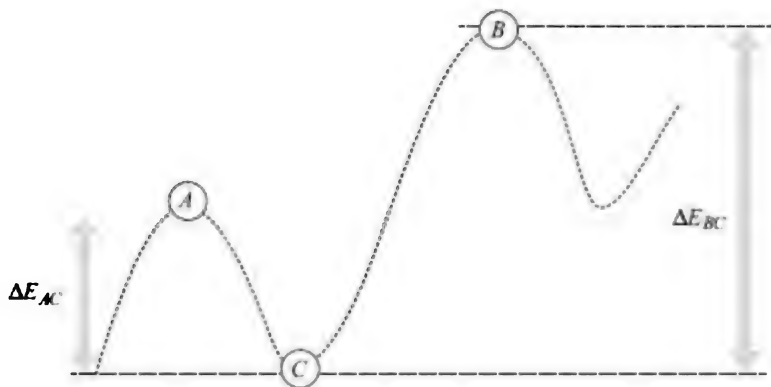


FIGURE 4.4 To move from *A* to *C*, the algorithm has to incur a smaller loss than to move from *B* to *C*. Simulated Annealing is more likely to move from *A* to *C* to *B* than vice versa.

One hopes, and this has been empirically supported by numerous applications, that by and large, the search will have a *tendency* to move to

better solutions. The search may not perform very well on surfaces like in Figure 4.2 where *IHC* worked well, but for jagged surfaces with many maxima, like the one shown in Figure 4.5 below, it will probably do well.

A randomized algorithm that has a simple and constant bias towards better values would be called *Stochastic Hill Climbing*. *Simulated Annealing* adds another dimension to this. It starts off being closer to the *Random Walk*, but gradually becomes more and more controlled by the gradient and becomes more and more like *Hill Climbing*. This changing behaviour is controlled by a parameter T called temperature. We associate high temperatures with random behaviour, much like the movement of molecules in a physical material. In *Simulated Annealing*, we start with a high value of T and gradually decrease it to make the search more and more deterministic. The probability of making a move at any point of time is given by a *sigmoid* function of the gain ΔE and temperature T as given below,

$$P(C, N) \leftarrow 1/(1 + e^{-\Delta E/T}) \quad (4.1)$$

where P is the probability of making a move from the current node C to the new node N , $\Delta E = (eval(N) - eval(C))$ is the gain in value and T is an externally controlled parameter called temperature. Note that the above formula is for maximization of the evaluation function value. For minimization, the negative sign in the denominator will be removed. The probability as a function of the two values is illustrated in Figure 4.6. The Y axis shows the probability value, and the X axis varies ΔE . The different plots are for different values of T .

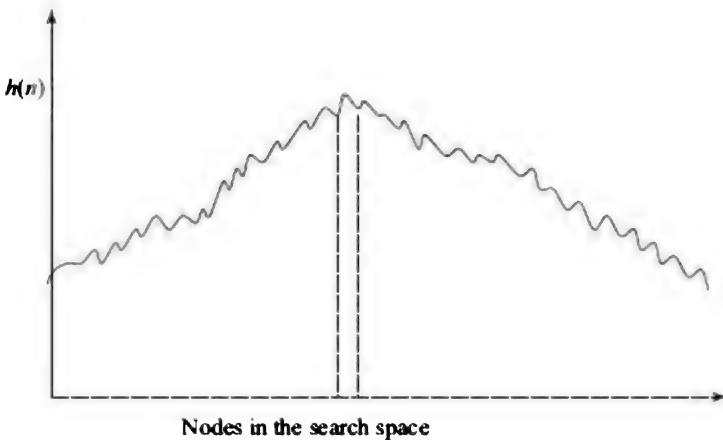


FIGURE 4.5 A jagged surface with many local optima, but a broader trend towards the optimum is well suited for *Simulated Annealing*. Observe that the footprint of *HC* is very small.

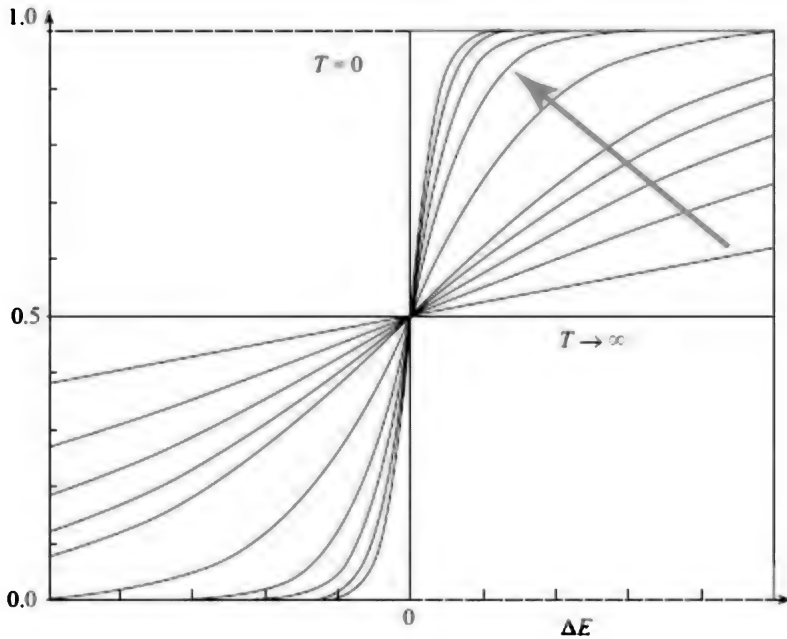


FIGURE 4.6 The probability curves (Sigmoid function) for different values of T . The probability of making a move increases as ΔE increases. For very large T , the algorithm behaves like *Random Walk*. As T tends to zero, the behaviour approaches the *Hill Climbing* algorithm.

When the temperature tends to infinity, the probability is uniformly 0.5, irrespective of the ΔE value. This means that there is equal probability of staying in the current node, and shifting to the neighbour. This is like the *Random Walk* algorithm. For finite values of T , one can see that when ΔE is positive, the probability is greater than half; and when it is negative, the probability is less than half. Thus, a move to a better node is more likely to be made, than a move to a worse node, though both have nonzero probability. For any given temperature, the greater the ΔE , the more the probability of the move being made. When ΔE is negative then the probability becomes lower as the loss in value becomes greater. Observe that for all temperatures, the probability is half when $\Delta E = 0$. This means that when the neighbour evaluates to the same value as the current node, it does not matter and one may or may not move to it with equal probability. Finally, one can observe that as temperature T becomes lower, the Sigmoid probability function tends to become more and more like a step function. When $T = 0$, it is a step function and the decision to move to the neighbour becomes a deterministic one. The search moves to the neighbour (with probability = 1) if the neighbour is better ($\Delta E > 0$), else it does not move to it (moves with probability = 0). This is like the *Hill Climbing* algorithm.

The *Simulated Annealing* algorithm does a large number of probabilistic moves with the temperature parameter being gradually reduced from a large initial value to a lower one. The manner in which the temperature is brought down is known as the *cooling schedule*. The intuition is that starting off with random moves that allow the search to explore the search space, the temperature is lowered gradually to make the search more exploitative of the

gradient information. As time goes by, the probability of the search being in the vicinity of the global optimum becomes higher, and at some point the gradient dominates the search and leads it to the optimum. The general outline of the SA algorithm is given below in Figure 4.7. There are many variations that can be done here, by choosing different cooling schedules. Another variation is one in which probabilistic moves are made only for negative gain moves.

```

SimulatedAnnealing()
1  node ← random candidate solution or start
2  bestNode ← node
3  T ← some large value
4  for time ← 1 to numberOfEpochs
5      do while some termination criteria /* M cycles in a simple case */
6          do
7              neighbour ← RandomNeighbour(node)
8              ΔE ← Eval(neighbour) - Eval(node)
9              if Random(0, 1) < 1 / (1+e-ΔE/T)
10                 then node ← neighbour
11                 if Eval(node) > Eval(bestNode)
12                     then bestNode ← node
13                 T ← CoolingFunction(T, time)
14 return bestNode

```

FIGURE 4.7 Simulated Annealing makes probabilistic moves in the search space. We use the function *Eval* instead of *h* in the style used in optimization. Function *CoolingFunction* lowers the temperature after each epoch in which some probabilistic moves are made. Function *RandomNeighbour* randomly generates one successor of the current node, and *Random(0,1)* generates a random number in the range 0 to 1 with uniform probability.

One way that the SA algorithm is different from the many algorithms we have seen is that it *does not* do local optimization. It does not look around the neighbourhood of a node for the best neighbour. Instead, it generates one successor or neighbour randomly, and then decides probabilistically whether to move to it or not. Thus, it can easily be used in problems where there are a large number of neighbours for a given node. In domains where one is dealing with real-valued variables, there could potentially be an infinite number of neighbours, like on a real hill; but this would not deter one from writing a search algorithm.

4.3 Genetic Algorithms

The natural world around us is a manifestation of life. In the world, and including this world itself, “*things that persist, persist and things that don’t, don’t*” (Grand, 2001). We can say that life is made up of forms that persist. We can also say that the goal of life is *persistence*, that is, to exist, or to live. Whichever way we look at it, life forms or living creatures are manifestations of different strategies to persist. Each *species* of life forms represents a strategy in which the *individuals* strive to persist. All living creatures have finite lifespans, but life itself continues as individuals beget offspring, which carry forward the strategy to live.

Persistence of living creatures requires energy, and different mechanisms have evolved to consume resources to generate the energy that sustains life.

Plants feed on the energy in sunlight and nutrients in the soil; rabbits, cows and goats eat the plants; lions, foxes and tigers eat the rabbits, cows and the goats. Living creatures die naturally too, and are consumed by bacteria, which in turn produce the nutrients for plants. Life exists in an ecosystem, in which a multitude of individuals from different species exist in a dynamic equilibrium (see Figure 4.8).

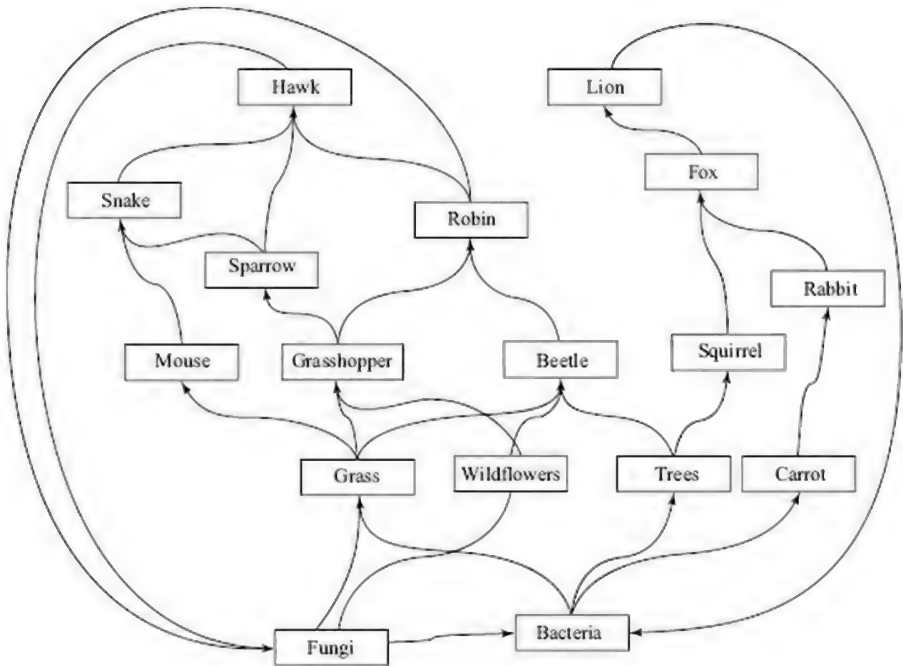


FIGURE 4.8 A small fragment of the vast ecosystem. The natural world contains millions of species interacting with each other. Arrows depict a positive influence of the population of one species on another.

Individuals may be born or may die, but life goes on. Even the extinction of a species or two does not stop this juggernaut. Once in a while though, some catastrophic event does occur that wipes out a part of the ecosystem and radically shifts the equilibrium, for example the one that happened about 65 million years ago wiped out the dinosaurs on Earth and created the opportunity for mammals to come to the fore.

Each species has its own strategy to live and consume resources, and when it can do that successfully, it survives. A species survives if the individuals of that species survive, at least long enough to create the next generation. Darwin called it the '*survival of the fittest*' (Darwin, 1859). A good strategy, a good design of a living creature, a *candidate solution for the problem of life*, is the one that survives. Life cannot explode limitlessly because the amount of matter is limited, and life uses matter to build creatures that live. And these creatures have to survive on the limited amount of energy that is available. So there is competition for survival, and life forms evolve by a process of natural selection. The ones that survive are the "fit" ones or good ones. The bad designs die out. In that sense, we can think of

nature as searching through different designs of living creatures.

While there is competition *between* the species, there is also competition *within* the species, perhaps more so. It is *this* competition that has led to the continual improvement of the species as a whole. The American paleontologist Edward Drinker Cope hypothesized that the body size of the members of a species grows with evolution (Cope's rule)³. This continues till a point when the members become so large, like the dinosaurs, that they become susceptible to rapid changes in environment, and become prone to extinction. It has been suggested that it is this competition that has resulted in the development of human brains and intelligence.

The species evolves and the ecosystem selects the individuals that survive longer. The species are interdependent upon each other. It is the individuals that participate in this interaction between the species. Each individual member strives for survival and propagation, and competes with other individuals for the means to do so. The fastest foxes catch the rabbits; the slower ones have to starve. The faster rabbits escape and live to eat and procreate. The early bird gets the worm. Life forms embody strategies that are more complex than just speed or timing. Foxes and rabbits may be clever too. Fruits and flowers rely on taste and looks for their seeds to be dispersed. Many predators and prey adopt disguises to hide from the other. In general, the strategies are complex and multi-faceted. Some simple strategies like that of a cockroach have survived for long periods, while other more complex ones like that of the cheetah seem endangered in this age of human dominance.

The offspring of a single individual would be like the individual in its strategy for survival. Nature has struck upon a novel way to produce improved designs. Procreation in nature is mostly through bisexual reproduction. Every species has male and female members, who mate and produce offspring. Selection happens because individuals that are alive and can find mates will get to reproduce, and others will not. When mating and reproduction does happen, the offspring *inherits features from both parents*. This process has the elements of a *random move* built in. Some children will inherit good features from both parents, and will be better than both, and will probably have better chances of survival and reproduction. In this way, "fitter" members of a species will survive and the population will have more and more fitter members.

The design of an individual is expressed in the genetic make-up of the individual, known as its *genotype*. A *chromosome* is a large macromolecule into which *DNA* is normally packaged in a cell and which contains many genes. The genes carry instructions encoded in a physical form in the *DNA* of an individual. Deoxyribonucleic acid (*DNA*) is a nucleic acid that contains the genetic instructions for the biological development of a cellular form of life or a virus⁴. *DNA* consists of a pair of molecules, organized as strands running start-to-end and joined by hydrogen bonds along their lengths. Each strand is a chain of chemical "building blocks", called nucleotides, of which there are four types: adenine (A), cytosine (C), guanine (G) and thymine (T). *DNA* is thought to have originated approximately 3.5 to 4.6 billion years ago, and is responsible for the genetic propagation of most inherited traits. In humans, these traits range from hair colour to disease susceptibility. The genetic

information encoded by an organism's *DNA* is called its *genome*. During cell division, *DNA* is replicated, and during reproduction is transmitted to offspring. The offspring's genome is a combination of the genomes of its parents. The *phenotype* of an organism, on the other hand, represents its actual physical properties, such as height, weight, hair colour, and so on. It is the physical entity or the phenotype that goes out and competes in the world. So while the genotype represents the inherited design, the phenotype is the individual that is the result of that design.

A newborn individual is like a computer program let loose in a system that can interpret it. Simple life forms, like the paramecium, earthworm, jellyfish or a cockroach, embody simple programs or strategies. More complex forms, like a typical mammal, embody more complex strategies. "Intelligent" life forms deploy very flexible strategies⁵. And in each species, the competition within has led to a continuous improvement in the strategy, through a process of natural selection.

Genetic Algorithms (GA) are a class of algorithms that try and build solutions by introducing evolution and selection of the best in a population of candidate solutions. The first thing one must do is to encode the problem by devising a schema for candidate solutions. One can think of this as an artificial chromosome, or the *DNA* sequence that is the blueprint of the solution. The chromosome is made up of the different components that make up the solution. Starting with a population of strings, GAs involve churning of the genes in search of better combinations.

There are three basic operations in *Genetic Algorithms* (Goldberg, 1989):

1. Selection

The selection operator allows the fitter candidates to be chosen for reproduction, and thus propagation of the genes (components). While in nature, the phenotype is out in the world competing for survival, such an approach is not suitable because the purpose of writing the GA is to produce a good design (solution). It would be too time consuming to build the phenotypes and test them in the real world, though such an approach has been used where simulation is possible (Sims, 94). In practice, GAs employ a user specified function to decide which designs/solutions are good or not. This function is called the *fitness function*, and gives a fitness value to each candidate. In optimization, we called this function the *evaluation function*. The selection operator takes a population of candidates and reproduces (clones) them in proportion to their fitness values. Fitter candidates will have more copies made, and the worst ones may not get replicated.

2. Recombination

The recombination operator takes the output of the selection operator, and randomly mates pairs⁶ of candidates in the population, producing offspring that inherit components (genes) from both parents. The new population thus contains a totally different set of solutions. In order to preserve the best candidates, sometime *elitism* is followed, that allows the best few candidates to have cloned copies in the new generation.

3. Mutation

Once in a while in the real world, a mistake happens and a child gets a mutated copy of a gene from the parent. Most of the time this happens, it is disastrous for the individual, but on a rare occasion the mutated gene is beneficial, giving rise to a more successful individual. GAs incorporate mutations to allow for the possibility of making good random moves. As a result, even when the population does not contain a good gene, there is a chance that it may arise out of a random mutation.

A typical framework of a *Genetic Algorithm* is given below. The most commonly used operator to recombine the genes from the two parents is known as the *crossover* operator. A *single point crossover* simply cuts the two parents at a randomly chosen point and recombines them to form two new solution strings. For example, if the solution has eight components then given the two parents:

$$\begin{aligned}P_1 &= X_1X_2X_3X_4X_5X_6X_7X_8 \\P_2 &= Y_1Y_2Y_3Y_4Y_5Y_6Y_7Y_8\end{aligned}$$

If we choose a crossover point between components 5 and 6, we get the two children as follows,

$$\begin{aligned}C_1 &= Y_1Y_2Y_3Y_4Y_5X_6X_7X_8 \\C_2 &= X_1X_2X_3X_4X_5Y_6Y_7Y_8\end{aligned}$$

A two point crossover would be equivalent to doing the above operation twice. One can define many crossover operators which will take components from the parents and mix them up. Care has to be taken that the resulting strings are valid candidates. This is easy when dealing with problems like SAT, where the i^{th} bit is a value of the i^{th} variable, but for most problems, the crossover point should be at a component level. Some problems like the *TSP* will not work with the crossover depicted above. We will look at *TSP* separately since it is a problem of considerable interest.

The algorithm GA is described at a high level in Figure 4.9.

In the algorithm shown in Figure 4.9, if we make the population size 1, and work only with the mutation operator then we would have the *Random Walk* or *Simulated Annealing* algorithms. The novel feature in GA is that it combines two processes working on a population of candidates. One recombines the components, and the other selects the best candidates. It is this dual action that makes GAs different. Without recombination, the algorithm would reduce to a parallel *Random Walk* or *Simulated Annealing*, where the mutation operator would be used to perturb solutions. *Perturbation* is a local change in the solution, and affects a local move in the neighbourhood graph. The crossover operator, on the other hand, results in a (big) jump in the neighbourhood graph. We can think of the crossover as a move from two parents to the two children. The two children may not be anywhere in the vicinity of the two parents, unless the two parents are similar to each other. At the same time, the *Selection* operator chooses the best (fittest) members and makes more copies of them. As the above two processes work in tandem, the population becomes fitter and fitter. This leads to more and more members accumulating near the peaks in the search space

(see Figure 4.10 below). If the number of peaks is small, this will in turn lead to a population that has more members that are similar to each other because they are all congregating near the peaks where the fitness values are high. Since more and more similar parents will be chosen to mate, the children will be closer to both of them. If there is only one peak, eventually the population will stabilise to most members becoming similar to each other and the genetic variation will diminish⁷, leading to fewer changes in future generations (Figure 4.11). This convergence behaviour is somewhat similar to the one in SA, which too starts off jumping about the search space.

```

GeneticAlgorithm()
1 Initialize an initial population of candidate solutions p[1..n]
2 repeat
3   Calculate the fitness value of each member in p[1..n]
4   selected[1..n] ← the new population obtained by picking n members
5     from p[1..n] with probability proportional to fitness
6   Partition selected[1..n] into two halves, and randomly mate and
7     crossover members to generate offspring[1..n]
8   With a low probability mutate some members of offspring[1..n]
9   Replace k weakest members of p[1..n] with the k strongest members of
10     offspring[1..n]
11 until some termination criteria
12 return the best member of p[1..n]

```

FIGURE 4.9 The GA algorithm works by reproducing a population in proportion to fitness, recombines the genes by crossover, and randomly mutates some members in each cycle. Parameter k decides how many of the parent population is to be retained.

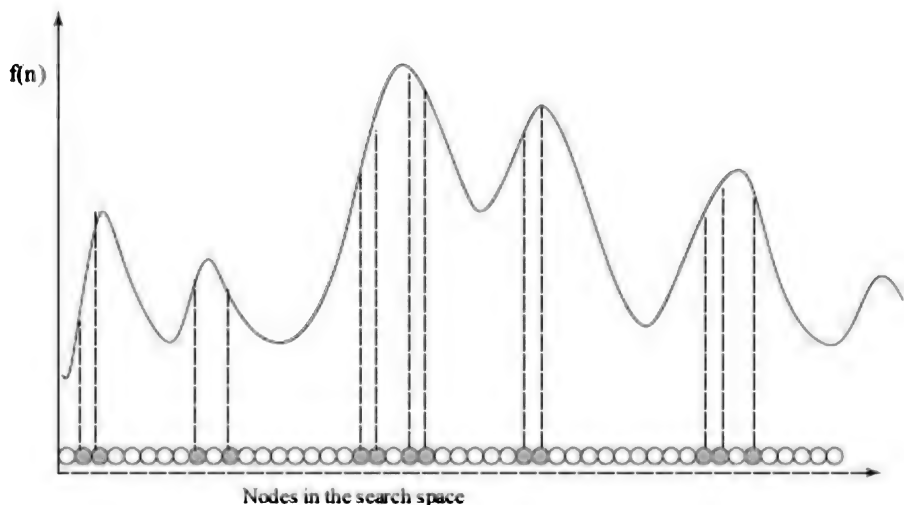


FIGURE 4.10 The Initial population may be randomly distributed, but as Genetic Algorithm is run, the population has more members around the peaks.

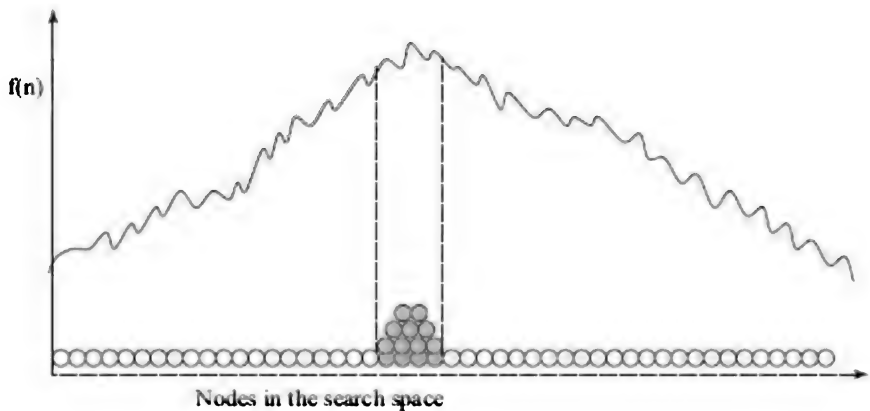


FIGURE 4.11 If there is one major peak, then most of the population is expected to converge towards that peak, leading to high similarity in the genetic make-up.

The genetic algorithms described here are instances of randomized processes at work. In particular, the mating and crossover is done by random selection from fit members. In the natural world, this process is far from random. Selection of mates is a conscious process in which attraction between opposite sexes plays a major role. One could say that both the concept of beauty and also its perception have *evolved* to help members choose mates that will result in better offspring. In a way then, that is a process of selection that operates at a different level. Another factor that affects mating is geographic location. Living creatures usually find mates from a nearby area. The structures of society and the often complex processes of matchmaking too, play a role. Many a time these are designed to serve a community or a clan, rather than the individual.

4.4 The Travelling Salesman Problem

The ‘*Travelling Salesman Problem*’ (TSP) is one of the harder problems around, and considered by some to be the holy grail of computer science. The problem can informally be defined as follows. Given a graph in which the nodes are locations or cities, and edges are labelled with the cost of travelling between cities, to find a cycle containing each city exactly once, such that the total cost of the tour is as low as possible. Thus, the salesman must start from a city, visit each city exactly once, and return home incurring minimum cost. The cost may be distance, time, or money. Most studies of TSP assume a completely connected graph. While practical problems, like a country map, generate graphs that are not complete, one can make them complete by adding edges with a very high cost.

Given N cities, a tour may be represented by the order in which the cities are represented, $(City_1, City_2, \dots, City_N)$ which is often abbreviated to $(1, 2, \dots, N)$. This is known as the *path representation*. One can observe that there are $N!$ permutations possible with N cities, each representing a tour. Every tour can be represented by N of these permutations, where each is a rotation of another. For example, in a nine city problem, $(1, 2, 3, 4, 5, 6, 7, 8, 9)$ represents the same tour as $(5, 6, 7, 8, 9, 1, 2, 3, 4)$. One must remember

percent of the estimated optimal solution was found in 8 CPU days on an IBM RS6000 machine (Applegate, 2003). More results on performance can be obtained from the Website for the DIMACS implementation challenge on *TSP* (Johnson et al., 2003) and (Cook, 2006). Some more interesting *TSP* problems available at (Cook, 2006) are: The *World TSP* -A 1,904,711-city *TSP* consisting of all locations in the world that are registered as populated cities or towns, as well as several research bases in Antarctica; *National TSP* Collection—a set of 27 problems, ranging in size from 28 cities in Western Sahara to 71,009 cities in China. Thirteen of these instances remained unsolved, providing a challenge for new *TSP* codes, and; *VLSI TSP* Collection—a set of 102 problems based on VLSI data sets from the University of Bonn. The problems range in size from 131 cities up to 744,710 cities.

TSP problems arise in many applications (Johnson, 1990), for example circuit drilling boards (Litke, 1984), where the drill has to travel over all the hole locations, X-ray crystallography (Bland and Shallcross, 1989), genome sequencing (Agarwala, 2000) and VLSI fabrications (Korte, 1990). These can give rise to problems with thousands of cities, with the last one reporting 1.2 million cities. Many of these problems are what are known as *Euclidean TSPs*, in which the distance between two nodes (cities) is the Euclidean distance. One can devise approximation algorithms that work in polynomial time. Arora (1998) reports that in general, for any $c > 0$, there is a polynomial time algorithm that finds a tour of length at most $(1 + 1/c)$ times the optimal for geometric instances of *TSP*, which is a more general case of an Euclidean *TSP*. Special cases of *TSPs* can be solved easily. For example, if all the cities are known to lie on the perimeter of a convex polygon, a simple greedy algorithm, *TSP-Nearest-Neighbour* shown below works.

4.4.1 Constructive Methods

Several constructive methods construct tours from scratch. We look at a few of them starting with the most intuitive nearest neighbour construction (Figure 4.12).

```

TSPNearestNeighbour()
1  currentCity ← some startCity
2  tour ← List(currentCity)
3  while an unvisited city exists
4      do neighbour ← NearestAllowedNeighbour(currentCity)
5          tour ← Append(tour, List(neighbour))
6          currentCity ← neighbour
7  return tour

```

FIGURE 4.12 The *Nearest Neighbour Tour* construction algorithm moves to the nearest neighbour (that has not been visited) at each stage. Function *NearestAllowedNeighbour* picks the nearest neighbour from the unvisited cities. The last segment back to the *startCity* is implicit.

The complexity of the algorithm is $O(bn)$, where n is the number of cities and b is the maximum degree of nodes (which is $n - 1$ for fully connected graph). We can observe that for most greedy algorithms for *TSP*, the complexity is $O(n^2)$. For *TSP* problems that satisfy the triangle inequality¹², it

has been shown that the tour found by the nearest neighbour algorithm is not more than $(\lceil \log_2(n) \rceil + 1)/2$ times the optimal tour cost (Rosenkrantz et al., 1977). In practice, the algorithm yields fairly good tours where there may be a few long edges that are added in the final stages. The reader can verify this by trying out the method on the problem shown in Figure 4.13. Observe that if the two extreme cities were not there, the cities would have satisfied the condition of being on a convex polygon, and the algorithm would have found the optimal solution.

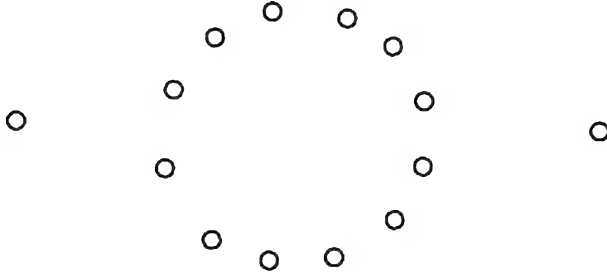


FIGURE 4.13 A near polygon arrangement of cities. The distances are the Euclidean distances in the figure.

We can improve upon the above algorithm slightly by extending the partial tour at the either end, choosing the one that has the nearest neighbour. The above algorithms work by adding nodes (cities) to the partial tours. The new node to be added is the one that is nearest to either end of the partial tour and extends the partial tour. Another heuristic method also adds nodes to the partial path, but selects the node to be added as the one that is nearest to *any node* in the partial tour. The new node is inserted into the partial tour by connecting it to the nearest node and splicing the new node. This is known as the *nearest insertion* heuristic, and it has been shown that the resulting tour costs are at most twice the optimal tour costs (Rosenkrantz et al., 1977). The reader is encouraged to try the heuristic on the problem in Figure 4.13.

We now look at constructive methods that add edges instead of nodes. The idea is that one adds edges making up the tour in some order, eventually stringing together the entire tour. In an algorithm known as the *Greedy Heuristic*, one starts off by sorting the edges in order of their costs (or weights or distances). The algorithm begins with the graph G containing all the nodes and no edges. One then keeps adding the cheapest edges to a partial tour being constructed in the graph G , such that no node ever has a degree more than two, and no cycles except the final tour exists.

Another method known as the *Savings Heuristic* starts off by constructing $(n - 1)$ tours of length two, all originating from a base node n_b . At the start, the i^{th} tour looks like (n_b, n_i, n_b) . The algorithm then performs $(n - 2)$ merges, at each stage removing two edges from two cycles to n_b and adding an edge to connect the two hanging nodes. Of the four combinations of edges that can be removed and replaced, the algorithm chooses the one that gives maximum *savings* in combined tour cost. Figure 4.14 below illustrates the algorithm.

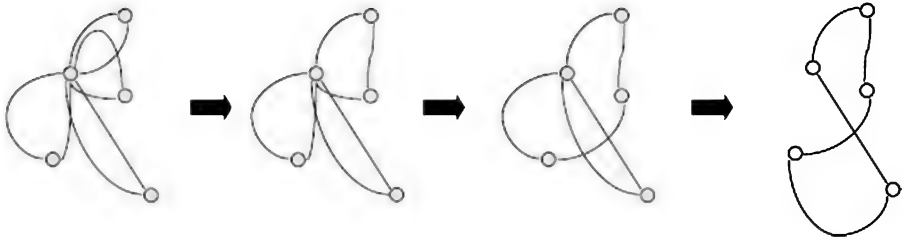
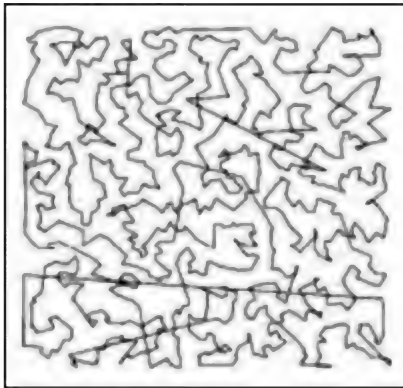
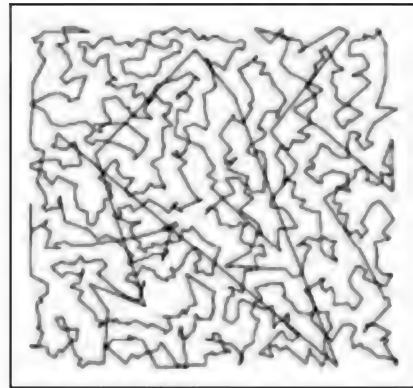


FIGURE 4.14 The *Savings Heuristic* starts with $(n - 1)$ tours of length 2 and performs $(n - 2)$ merge operations to construct the tour.

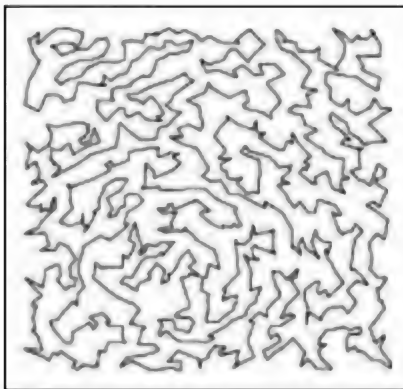
It has been shown (Ong and Moore, 1984) that the *Greedy Heuristic* produces tours at most $(1 + \log(n))/2$ longer than the optimal tour, and the *Savings Heuristic* at most twice of that. In practice, however, the *Savings Heuristic* has empirically produced better tours. The following tours from the DIMACS Webpage (Figure 4.15) illustrate typical performances of these heuristic algorithms.



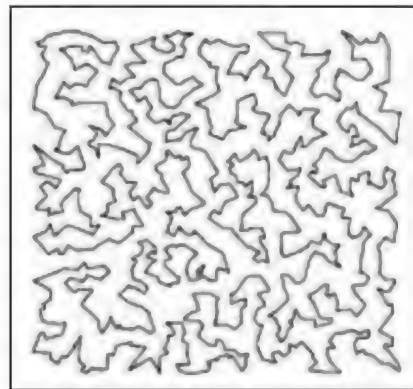
Greedy tour



Nearest neighbour tour



Savings tour



Optimal tour

FIGURE 4.15 Tours found by some heuristic constructive algorithms. Figure taken from <http://www.research.att.com/~dsj/chtsp/>

4.4.2 Perturbation Methods for the TSP

Search methods that operate in the solution space can be easily applied to *TSP* problems. They work on complete tours, and the neighbours of a search node are tours obtained by perturbing the given tour in some way. In the path based representation of a tour, a given tour can be modified by exchanging any two cities in the tour, as shown in Figure 4.16. For example, cities 4 and 5 can be exchanged in (2, 3, 4, 1, 7, 6, 5, 8, 9) to give (2, 3, 5, 1, 7, 6, 4, 8, 9). We can call this the 2-city-exchange move. For a tour of N cities, this generates NC_2 neighbours for any given tour. We can also exchange more than two cities to design a neighbourhood function k -city-exchange. Notice that if we remove 3 cities from the tour, we can put them back in $(3!-1) = 5$ other different ways, thus giving us ${}^NC_3 * (3!-1)$ neighbours for any given tour. In general, as k increases, the neighbourhood function becomes denser.

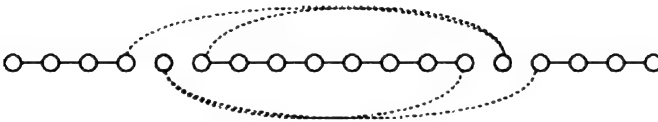


FIGURE 4.16 In the 2-city-exchange, the two shaded cities exchange their position in the path representation. The new tour has the dashed edges instead of the four broken ones in the linear order.

Another way of defining neighbourhood functions is to think of exchanging edges instead of cities in a tour. In a 2-edge exchange move, we would remove two edges and cross connect them to give a different tour, as shown in Figure 4.17. For example, in the tour (2, 3, 4, 1, 7, 6, 5, 8, 9), we could remove edges 4-1 and 8-9 and replace them with 4-8 and 9-1 to give a tour (2, 3, 4, 8, 5, 6, 7, 1, 9). The reader is encouraged to verify that this is the only way the two edges can be replaced to give a valid tour. It can also be observed that the 2-edge-exchange can be implemented by reversing a subsequence, in this case 1-7-6-5-8 in the path representation. We can do a 3-edge exchange by taking out and replacing three edges from the tour, and this can be done in four different ways (see Figure 4.18).

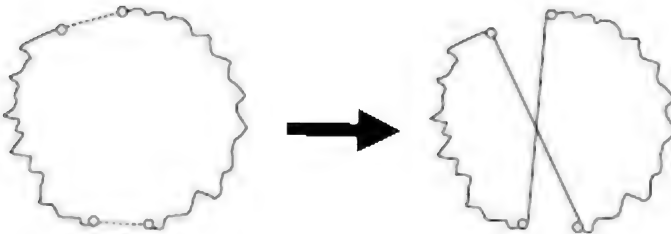


FIGURE 4.17 In the 2-edge-exchange, the tour is altered as shown.

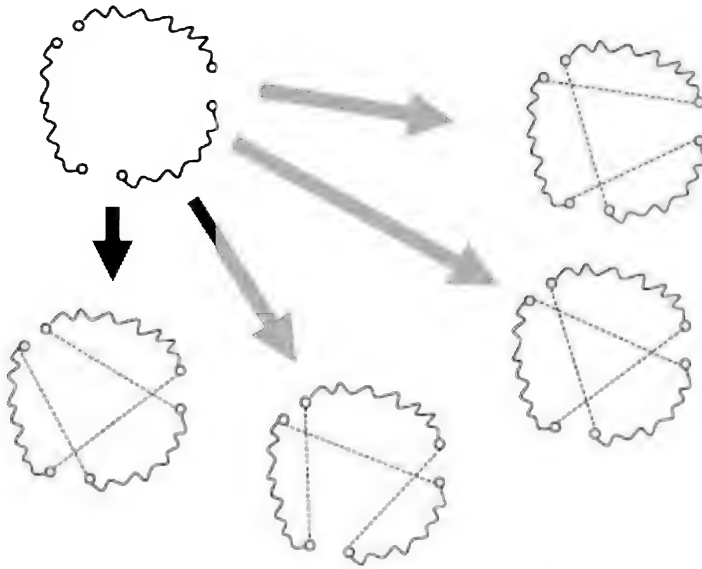


FIGURE 4.18 In a 3-edge-exchange, three edges are removed and replaced by other edges.

A closer look at the 2-city-exchange shows that it is one particular case of 4-edge-exchange in which two sets of two consecutive edges are removed from a tour, and each freed node is connected to the neighbours of the other freed node. Experiments have revealed that edge exchanges give better results than city exchanges. They are also intuitively more appealing. For example, one can visualise the 2-edge-exchange undoing a crossed path in a tour (if the move shown in the figure were reversed). It also allows for using a heuristic like selecting longer edges to be removed from a given tour. In general, using a higher k -exchange operator gives a denser neighbourhood. In the case where $k = n$, the entire search space comes in the neighbourhood, as perturbations are no longer local. The graph is fully connected as all permutations become neighbours. We have seen that a deterministic search method like *Variable Neighbourhood Descent* (Chapter 3) can exploit increasingly denser neighbourhood functions. The *TSP* problem, like the *SAT* problem, allows for a number of neighbourhood functions of increasing density, and one can implement algorithms that work within the capacity of a given set of resources.

4.4.3 GAs for TSP

Instead of perturbing a given solution, GAs generate new candidates by inheritance and recombination of solution components from two parents selected, based on their fitness. The idea is to serendipitously combine good components that occur in the two parents. In the *TSP* problem, the components are the individual segments in a tour, and also subtours made up a sequence of segments. For GAs to work well, one must be able to devise crossover operators that allow for such recombination. This is a somewhat difficult task, and different crossovers, and even alternate representations,

have been tried. We look at a few of them here.

Path Representation

In path representation, the simple one point or multipoint crossovers defined in this chapter earlier do not work because the resulting sequences are not likely to be valid tours. For example, given two parent tours

$$P_1 = (2, 4, 7, 1, 5, 9, 8, 3.6), \text{ and} \\ P_2 = (4, 5, 7, 2, 6, 8, 1, 9, 3)$$

if we do a crossover after four segments, we get the two offspring:

$$C_1 = (2, 4, 7, 1, 6, 8, 1, 9, 3), \text{ and} \\ C_2 = (4, 5, 7, 2, 5, 9, 8, 3.6)$$

Neither of the two offspring is a valid tour because cities repeat in them. We need crossover operators that will retain the n cities and only introduce a different order. Some interesting crossovers that have been tried out (Michalewicz and Fogel, 2004) are as follows.

The *Partially Mapped Crossover* (PMX) builds a child as follows. It chooses a random subsequence from one parent, and fills in the remaining tour by maintaining the order and position of cities as in the other parent. For the above example, choosing the subsequences from fourth to seventh city gives the children.

$$C_1 = (x, x, x, 1, 5, 9, 8, x, x), \text{ and} \\ C_2 = (x, x, x, 2, 6, 8, 1, x, x)$$

This defines a series of mappings ($1 \leftrightarrow 2$), ($5 \leftrightarrow 6$), ($9 \leftrightarrow 8$) and ($8 \leftrightarrow 1$). Next, the cities that can be copied from the other parent are brought in to give

$$C_1 = (4, x, 7, 1, 5, 9, 8, x, 3), \text{ and} \\ C_2 = (x, 4, 7, 2, 6, 8, 1, 3, x)$$

Finally, the four x 's in the above children are replaced by 6 (from the mapping ($5 \leftrightarrow 6$) because 5 should have been there), 1 (from the mappings ($9 \leftrightarrow 8$) and ($8 \leftrightarrow 1$), 9 (from ($1 \leftrightarrow 2$), ($8 \leftrightarrow 1$) and ($8 \leftrightarrow 9$) – it should have been 2, but since 2 is already there 1, but since 1 is also there 8, and since 8 is already there 9), and 5, to give the two offspring

$$C_1 = (4, 6, 7, 1, 5, 9, 8, 6, 3), \text{ and} \\ C_2 = (9, 4, 7, 2, 6, 8, 1, 3, 5)$$

In *Order Crossover* (OX), we copy a substring from one parent as in PMX, but fill in the remaining nodes in the order they occur in the other parent. Starting with

$$C_1 = (x, x, x, 1, 5, 9, 8, x, x), \text{ and} \\ C_2 = (x, x, x, 2, 6, 8, 1, x, x)$$

the remaining cities for C_1 are arranged in order as in $P_2 = (4, 5, 7, 2, 6, 8, 1, 9, 3)$ giving $(4, 7, 2, 6, 3)$.

Likewise for the other child, the remaining cities are arranged in the order of P_1 giving the two offspring

$$C_1 = (1, 5, 9, 8, 4, 7, 2, 6, 3) \text{ and} \\ C_2 = (2, 6, 8, 1, 4, 7, 5, 9, 3)$$

In *Cycle Crossover* (CX), the attempt is made to inherit the position of each city in the offspring from one of the two parents as far as possible. We start constructing the two offspring with the first city as

$$C_1 = (2, x, x, x, x, x, x, x, x), \text{ and} \\ C_2 = (4, x, x, x, x, x, x, x, x)$$

Since CX tries to inherit the position of each city from one of the parents, it can be seen that in C_1 the position of city 4 can only be inherited from P_1 , because the first slot where 4 occurs in P_2 is already used up in C_1 . Likewise, the position of city 2 in C_2 can only be inherited from P_2 , giving

$$C_1 = (2, 4, x, x, x, x, x, x, x), \text{ and} \\ C_2 = (4, x, x, 2, x, x, x, x, x)$$

Now since 4 is in second place in C_1 , the position of the other city in the second place, that is city 5, can only be inherited from one parent, which is C_1 . In this manner, we propagate the constraints till we cannot fill up a city without creating a cycle in the partial tour. At this stage, we have

$$C_1 = (2, 4, x, 1, 5, 9, 8, 3, 6), \text{ and} \\ C_2 = (4, 5, x, 2, 6, 8, 1, 9, 3)$$

The remaining cities, in this case there is only one, is filled up from the other parent. Let us look at another example where a smaller subtour is selected. Given two parents

$$P_3 = (1, 2, 3, 4, 5, 6, 7, 8, 9), \text{ and} \\ P_4 = (3, 4, 6, 5, 2, 7, 9, 8, 1)$$

the reader should verify that the subtours chosen by from one parent are

$$C_3 = (1, x, 3, x, x, 6, 7, x, 9) \text{ from } P_1, \text{ and} \\ C_4 = (3, x, 6, x, x, 7, 9, x, 1) \text{ from } P_2$$

In the positions marked x, the cities from the two parents are exchanged, giving

$$C_3 = (1, 4, 3, 5, 2, 6, 7, 8, 9), \text{ and} \\ C_4 = (3, 2, 6, 4, 5, 7, 9, 8, 1)$$

We can see the cycle crossover as a carefully chosen simple multipoint crossover, in which selected parts of the two “chromosomes” are exchanged.

Ordinal Representation

Interestingly, there does exist a tour representation where the simple crossover can be used producing valid tours. This is known as the ordinal

representation. We begin by arranging the cities to create a reference order R of the cities. Since we are interested in the index of cities in this reference, we name the cities themselves by letters to avoid confusion.

$$R = (A, B, C, D, E, F, G, H, I)$$

Then a tour, say C-D-F-E-B-G-I-H-A, is represented as a list L of references which is constructed as follows. The first entry in L is the reference index from R for the first city in the tour, in this case, city C in position 3 in R . We also modify the reference index R by deleting the city C from it.

$$L = (3)$$

$$R = (A, B, D, E, F, G, H, I)$$

The next city in the tour D is now in the third position in the modified reference R . We have,

$$L = (3, 3)$$

$$R = (A, B, E, F, G, H, I)$$

The next city F is in the fourth position in the updated R .

$$L = (3, 3, 4)$$

$$R = (A, B, E, G, H, I)$$

Continuing in this manner, we get a tour representation for the given tour as,

$$L = (3, 3, 4, 3, 2, 2, 3, 2, 1)$$

The reader is encouraged to verify that with the ordinal representation, one can use an arbitrary simple crossover operator that will yield valid tours as offspring. The advantage is that while the coding of tours may take some computational effort, once we have a population of tours available, the GA implementation becomes faster because we can use, say a single point crossover.

Adjacency Representation

Another representation of tours that has been experimented with is the adjacency representation. This is an indexed representation in which city i is in position j in the list if in the tour the salesman goes from city i to city j . For example, the representation (4, 5, 7, 2, 6, 8, 1, 9, 3) can be interpreted as follow. Starting with 1, the next city in the tour is 4, because 4 is in the first place in the representation. From 4, the next hop is to city 2, because 2 is in the fourth position. The complete tour can in this manner be constructed as 1-4-2-5-6-8-9-3-7-1.

One problem with the above representation is that not every permutation of cities represents a valid tour. For example, no permutation can start with 1, because that would mean going from 1 to 1. Furthermore, any permutation that contains say (3, x, 5, x, 1, x, x, x, x), is not a tour because it contains a cycle 1-3-5-1. Also, the single point crossover will not work. The appeal of the representation lies in the fact that it represents explicitly where to go from any

given city. Thus, we can say in the tour (4, 5, 7, 2, 6, 8, 1, 9, 3) that after city 3 one goes to city 7, because 7 is in the third location in the representation. Given two parents, in the adjacency representation, the two options in the two parent tours are available at the same location, and could thus be inherited from either parent. One could then construct a child tour by choosing the next location to go to, using some heuristic approach.

In the *Heuristic Crossover* (HX), a child tour is constructed by choosing a random city as a starting point. The next city is chosen from the two options in the two parents, by choosing the one that is linked by a shorter edge. One has to keep a lookout for the formation of a cycle in the tour, and if that is happening at some stage, a random city is chosen that does not introduce a cycle. Observe that if one of the parents has a sequence of cities connected by short edges, they are likely to be carried forward to the offspring being constructed.

One can simplify the above crossover by choosing the successor cities from the two parents alternately. This is known as the *Alternating Edges Crossover* (AEX). A variation of this is to select a sequence of edges from one parent before choosing some from the other. This is known as the *Subtour Selection Crossover* (SSX). One can observe that this is similar to the PMX crossover with path representation. A check for cycle formation has still to be kept in all these approaches though.

In summary, the *TSP* problem is one of those problems in which defining crossover functions is not a straightforward process. The intuition behind GAs is that the offspring have a chance of inheriting and combining good components from the parents. In *TSP*, the good components are tour segments. But the shortest tour segments may not add up to a valid tour. If this happened, a greedy tour construction would have worked.

Box 4.1: A Note on Innovation and Creativity

Over a period of 3.5 billion years, nature has produced a vast plethora of designs for life through a process of recombination and selection. Starting presumably with a few simple forms, life has diversified into a number of species, with a large number of individuals within the species. Life forms display a bewildering diversity, occupying all nooks and corners of the earth. The number of living creatures living can safely be said to be unknown. Even the number of species is not known, and is variously said to be anywhere between 3 to 100 million. The world around us is a result of billions of generations of recombination and selection over a population of billions.

In contrast, human endeavour is relatively short termed. Our problem solving efforts are focused on speed and minimal use of resources. There has been considerable evidence that the completely unconstrained methods employed by nature are too slow to solve our problems (Grand, 1998; Goldberg, 2002). If we are to devise GAs that work, we have to put in more structure to guide the search. (Goldberg, 2002) defines *competent* GAs as follows.

"The primary objective is to design what we call competent genetic algorithms. A GA is called competent if it can solve hard problems

quickly, accurately, and reliably. Each of these can be quantified further, but qualitatively, hard problems are those that have large subsolutions that must be discovered whole, badly scaled subsolutions, many different local optima, a high degree of subsolution interaction, or a lot of external noise or stochasticity. In short, we are interested in designing effective solvers for the class of nearly decomposable problems (Simon, 1969). Speed, accuracy, and reliability requires that we get to near-global or high-quality solutions in times that grow as a polynomial function of the number of decision variables with high probability."

The competent GA is centred around decomposing problems into subproblems and finding subsolutions of those subproblems and combining them to "build" the solution. The subsolutions are called *Building Blocks* (BB). This is clearly different from the GAs described in this chapter because it looks for ways to *find and preserve BBs*. Unconstrained GAs will take too long¹³ a time to do this. Clearly, the design of GAs that will find solutions in reasonable time is still in the realm of being an art form.

Having expressed the above caveat, we would like to observe that the dual process of *recombination and selection* is the only known approach to creativity. Moreover, reports on the study of human creativity have repeatedly thrown up the notion that innovation and creativity arise when humans combine and recombine ideas, often subconsciously; and have the ability to latch onto a good idea when it does emerge. We look at some of the evidence reported below.

The French mathematician Hadamard (1954) has attributed innovation and discovery to recombination in a pool of ideas "*We shall see a little later that the possibility of imputing discovery to pure chance is already excluded....Indeed, it is obvious that invention or discovery, be it in mathematics or anywhere else, takes place by combining ideas*". He also quotes the French poet Paul Valéry who argued, "*It takes two to invent anything. The one makes up combinations: the other chooses ... what is important to him in the mass of the things which the former has imparted to him.*" This statement describes the two processes of recombination and selection succinctly.

Recent work employing techniques from psychology and brain imaging has shown that a key ingredient to creativity is the ability to handle many diverse kinds of ideas together, allowing them to cross fertilize each other. Our normal thinking processes are tuned to focusing on relevant matters and ignoring information that may be irrelevant to the task at hand. This process of ignoring irrelevant inputs is called *latent inhibition*. With high levels of latent inhibition, a person can pursue a task with a single mindedness that is a highly valued trait. The work done in Harvard University by a team led by Shelley Carson, a Harvard psychologist, reports a study that reveals that people with low levels of latent inhibition are the ones who can combine ideas creatively (Carson, 2003). Interestingly, low levels of latent inhibition are also related to psychotic conditions. "*Scientists have wondered for a long time why madness and creativity seem linked, particularly in artists, musicians, and writers. ... Our research results indicate that low levels of latent*

inhibition and exceptional flexibility in thought predispose people to mental illness under some conditions, and to creative accomplishments under others.” (Carson, 2003, Cromie, 2003).

Unlike the “normal” thought processes that are focused, people with low levels of latent inhibition have their minds flooded with many different ideas. If the person is able to handle this meaningfully, she can be creative; otherwise there is a danger of being labelled as ‘mad’. Low latent inhibition, it seems, increases the available “*mental elements*”—thoughts, memories, and the like, or what Carson calls “*bits and pieces in the cognitive workspace*”—that supply the raw material for originality and novelty. Although too much material entering the “*cognitive working area*” might disorient psychotics, Carson wondered whether “*highly creative people could use those many bits and pieces in the cognitive workspace and combine them in novel, original ways.*” (Lambert, 2004). “*Perhaps they do not dismiss as easily as the rest of us “irrelevant” ideas that pop into their heads, but instead entertain them long enough for one of them to connect with another thought that is kicking around—giving birth to a novel, creative idea. ... Getting swamped by new information that you have difficulty handling, may predispose you to a mental disorder,*” Carson says. “*But if you have high intelligence and a good working memory, you are more likely to be able to combine bits of new information in creative ways.*” The “*high intelligence*” part of Carson’s statement is key (Begley, 2005). High intelligence, she adds, “*should help you to better process the increasing information that goes along with low latent inhibition. To be creative, you can be bright and crazy, but not stupid.*” Some minimal level of intelligence is, therefore, required for creativity. The reason is that in order to generate novel combinations, it helps to have a wealth of mental elements to work with. Without a *sufficient supply* of elements that can be combined in an original way, creativity is impossible (Begley, 2005).

This view is also expressed by Dean Simonton, a psychologist from the University of California, Davis. He says (Simonton, 2004) that creativity is analogous to variation and selection in Darwinian evolution. “*The creator must generate many different novelties from which are selected those that satisfy some intellectual or aesthetic criteria. ... Underlying creativity, therefore, must be some process that generates these variations, made up of novel combinations of cognitive bits and pieces, as well as some way to choose among them. Creative people in diverse fields have said that this is exactly what it feels like they did. Chemist Linus Pauling described the need to “have lots of ideas and throw away the bad ones.... You aren’t going to have good ideas unless you have lots of ideas and some sort of principle of selection.” Mathematician Henri Poincaré recalled the feeling that accompanied a creative breakthrough: “Ideas rose in crowds; I felt them collide until pairs interlocked, so to speak, making a stable combination. By the next morning, I had established the existence of a class of (previously unknown mathematical) functions.” Einstein described how “combinatory play seems to be the essential feature” in creativity (Begley, 2005).*

“*One eternal question is the relationship between madness and*

creativity. To be sure, most people who are mentally ill are not especially creative. But history is full of creative geniuses who were insane, including Vincent van Gogh and Robert Schumann; those who committed suicide, such as Ernest Hemingway and Virginia Woolf; or who were paranoid, such as Sir Isaac Newton. In the largest study ever conducted of the connection between creativity and madness, Arnold Ludwig analysed the biographies of about one thousand eminent men and women (Ludwig, 1995). He found that mental illness occurred more frequently in this group than it did in the general population. Specifically, 60 percent of the composers had psychological problems, as did 73 percent of the visual artists, 74 percent of the playwrights, 77 percent of the novelists and short-story writers, and 87 percent of the poets. But only about 20 percent of scientists, politicians, architects, and business people had even mild mental illness.” (Begley, 2005).

It is the pathways and flexible connectivity in the human brain that determines what thought patterns can come together to “mate” and produce novel ideas.

Can we implement systems that are creative? The above discussion indicates that creativity involves the ability to make connections between seemingly unrelated ideas and selecting the useful combinations. Our exploration of GAs has just begun. The chromosomes we talk about are all of strings of the same length and a fixed schema. Though there has been some research with variable length strings and working with tree structures (see (Koza, 1992)), we are still far away from having integrated representations for all the problems our agents solve. As a result, each system that we develop for a specific task has its own representation. We will see in Chapter 15 that repositories (populations) of solutions can also be accumulated and used in the form of experience by memory-based agents. Even here, the schema of the solution is fixed and each system is designed for a specific task. Building integrated knowledge representation systems that an agent can use for different tasks is definitely a challenge for artificial intelligence research. Until we can do so, there is little chance of being able to make “long-distance” cross connections that seem to be the source of novel solutions. In some sense, our systems are tuned to work within a species, making changes within a rigid framework. To mimic, natural evolution would also result in accepting the slow process of change that nature has worked with. To emulate human creativity, we must be able to represent diverse ideas in a common pool and develop an ability to combine them in different ways and recognizing the good ideas when they do come along.

Box 4.2: Swarms in the Semiosphere

In genetic algorithms, we talk about systems (species, designs and solutions) that are made up of parts; and of processes that recombine the parts to produce new and novel systems. A question one might ask is what these processes are and how these parts come together. The

idea of emergent systems is that these parts come together by themselves in an environment where *"things that persist, persist and things that don't, don't"* (Grand, 2001). The idea behind *emergent systems* is that small simple parts *can* come together to form systems that *can* display complex behaviour.¹⁴ And that simple parts come together and cooperate; not as a conscious process, but because their simple behaviours mesh together easily. Well known examples of complex systems emerging out of combinations of simple ones are ant colonies and human brains.

Jesper Hoffmeyer says¹⁵, *"The emerging discipline of biosemiotics looks at how complex behaviour emerges when simple systems interact with each other through signs. Sign processes (or semiosis) are processes whereby something comes to signify something else to somebody (and 'somebody' here may be taken in the broadest sense possible, as any system possessing an evolved capacity for becoming alerted by a sign). The study of living systems from a semiotic (sign theoretic) perspective is called biosemiotics. According to biosemiotics, most processes in animate nature at whatever level, from the single cell to the ecosystem, should be analysed and conceptualized as sign processes. Biosemiotics is concerned with the sign aspects of the processes of life. In the biosemiotic conception, the life sphere is permeated by sign processes (semiosis) and signification. Whatever an organism senses also means something to it, food, escape, sexual reproduction, etc.; and all organisms are born into a semiosphere. The study of signs is known as semiotics, and the notion of a semiosphere (Lotman, 1990) refers to a world of meaning and communication: sounds, odours, movements, colours, electric fields, waves of any kind, chemical signals, touch, etc. The semiosphere poses constraints or boundary conditions to the Umwelts of populations since these are forced to occupy specific semiotic niches, i.e. they will have to master a set of signs of visual, acoustic, olfactory, tactile and chemical origin in order to survive in the semiosphere."*

The behaviour of termites and ants has long been understood as being coordinated through a system of signs. Higher level creatures like mammals routinely employ signs for communication. Many animals are known to mark their territory through various means. Humans, of course, have taken communication through symbols to a totally new level with the invention of language. Even outside of language, we have a rich and diverse system of communicating through signs, and by doing so, we often (consciously) organize ourselves into teams and mobs where the activity of one becomes a sign for others to interpret. One has only to watch a high level team sporting event, like soccer, to see how the players "read" the actions of their teammates as well as opponents. We have no difficulty in accepting the fact that mammals and humans communicate through signs. Biosemiotics however, shows that communication through signs may result in simpler entities coming together, and forming swarms that can be seen as a more complex system. Consequently, it emerges that not only are our social structures immersed in a semiosphere, but also that we ourselves *are* kinds of

semiospheres, in which the cells making up our different organs are bound together.

Hoffmeyer (1994) introduces a notion of *semiotic interaction* (from Greek: *semeion* = sign, *etos* = habit) interaction between simple elements as a general phenomenon in the life sphere. Semiotic interaction refers to the tendency of living systems to make signs based on any persistent regularity: wherever there has developed a *habit*, there will also exist an organism for whom this habit has become a sign. He illustrates this with the behaviour of termites in the following paragraph —“*When termites initiated nest constructing, the following sequence of events was observed by Grassé: First, hundreds of termites move around at random, while they exhibit a peculiar habit of dropping small pellets of masticated earth in places which are elevated a little bit from the ground. In spite of the disorganized character of this activity, it results in the formation of small heaps of salivated earth pellets. Second, these heaps of earth pellets are interpreted by the termites as a sign to release a new habit. Every time a termite meets a heap, it energetically starts building earth pellets on top of it. The effect of this activity will soon be the formation of a vertical column. The activity stops when the column has reached a certain species-specific height. Third, if the column has no immediate neighbours, the termites completely stop bothering about it. But if in an adequate distance there are one or more other columns, a third habit is released. The termites climb the columns and start building in a sloping direction towards the neighbouring column. In this way, the columns become connected with arches. The amazing fact is that through a seemingly haphazard sequence of events, a nest is actually produced which cannot but elicit the feeling in the observer, that there must have been some kind of intelligence behind it.*”

The last statement resonates with Richard Dawkins who says that nature is “*The Blind Watchmaker*” who has fashioned the world. Looking at life from the perspective of a semiosphere, one can even think of the living forms achieving persistence by passing information about themselves in their genes. The notion is best described in another quote from the author (Hoffmeyer, 1994a)—“*To grasp the fundamentally semiotic character of animate nature, let us begin by considering the key process in life’s peculiar way of persistence, heredity. Heredity is a phenomenon which is now rather well understood. And yet its real significance is rarely properly explained. The significance is this: Since living systems are mortal, their survival has to be secured through semiotic rather than physical means. Heredity is semiotic survival, i.e. survival through a message contained in the genome of a tiny template cell, the fertilized egg in sexually reproducing species. ... In addition to this vertical semiotic system, i.e. genetic communication down through the generations, all organisms also partake in a horizontal semiotic system, i.e. communication throughout the ecological space. Every organism is born into a world of significance. Whatever an organism senses also mean something to it, food, escape, sexual reproduction, etc. This is one of the major insights brought to light through the pioneering work of Jakob von Uexküll: “Every action, therefore, that*

consists of perception and operation imprints its meaning on the meaningless object and thereby makes it into a subject-related meaning-carrier in the respective Umwelt (subjective universe)" (Uexküll, 1982).

One can then imagine all activity in terms of hierarchically composed semiotic systems, in which smaller parts come together (by themselves, through a long process of experimentation of recombination and selection) to form bigger and bigger "entities" that are more complex than the parts that make them up. The swarms like those of bees and ants may be made up of components that are similar, as are human organizations.

It must be observed though that even when *similar* members organize themselves into larger entities, they take up different roles. A typical corporate house, for example, will have CEOs, directors, managers, engineers, technicians, drivers, typists, cleaners, accountants and even lawyers. So they are similar as human beings, but functionally very different. In the same manner, we can think of our own body as made up of functionally different parts, which have some basic level of similarity, in that they all contain the same genetic code. One must keep in mind though that this analogy is a very loose one since bodies and industrial houses differ in many other ways. The point is that maybe one can loosen the strong sense of focused "self" that we all have and try and visualise the creation of life forms as something that arises out of a bottom up process as described by Hoffmeyer (Hoffmeyer, 1994) —*"The general principle which has made this bottom up or swarm conception of the body-mind biologically possible is the introduction of semiosis as the basic principle of life. By delegating semiotic competence to decentralised units, and ultimately to single cells, it becomes possible to ascribe intelligent behaviour to distributed systems. Stupid molecules become powerful tools as soon as they acquire semiotic quality, i.e. as soon as they are interpreted according to cellular habits. The transformation of molecules to signs opens for an unending semiogenic evolution based on semiotic interaction patterns between entities at all levels. And through this evolution, the semiotic aspects of material processes gradually increase their autonomy, thereby creating an ever more sophisticated semiosphere. A semiosphere which finally had the power to create semiotic systems, such as thoughts and language, which are only in the slightest way dependent on the material world from which they were ultimately derived."*

4.5 Neural Networks

The discipline of biology has revealed that all life forms we see around us are *colonies of cells* that exist in a symbiotic equilibrium.

Life exists in many forms varying from the simple organisms made up from a few cells to very complex creatures hosting hundreds of *types* of cells. From the simple amoeba to the human being, there is an increase in complexity of behaviour accompanied with the faculty of awareness. Somewhere along this order of increasing complexity, the *notion of a self* emerges and creatures like

us can think about ourselves. The real wonder though is that *we can have a notion* at all! Creatures can somehow think about things; about the world around them; and about themselves. The notion of self is possible only when notions are possible in the first place. And when notions about the self and the world become possible then creatures can act purposefully and intelligently, increasing the chances of their own survival.

Minds in bodies have, for long, confounded thinkers. Philosophers across cultures and times have grappled with the nature of reality and the relation between the mind, that perceives the reality, and the body, that is presumably real. Indian philosophers starting with Gotama (in 6th century BCE¹⁶), Vatsyayana (2nd century BCE), Vacaspati Misra (9th century CE) up to Madhusadana Sarasvati (16th century CE) have explored the notion of mind-body dualism (Chakrabarti, 1999). European philosophers too, including Thomas Hobbes (16th century), René Descartes (16th century) and David Hume, (18th century), have struggled to understand how we can have minds (Haugeland, 1985). *Idealism* says that everything is in the domains of minds, or ideas, and that matter is an illusion. The opposite more prevalent view, *materialism*, is that everything is matter and that mind is a construct of the activity in the physical brain.

Recent progress in biology, neuroscience and computer sciences have reinforced the notion that thinking is some kind of a process that happens in a body that emerges as a whole from the constituent parts. And the seat of thought is the brain. It has also been hypothesized that the mind itself emerges from a combination of simple mental faculties (Minsky, 86). And all this happens in the brain.

The human brain has been described as the “*most complex piece of matter in the universe*” (Ramachandran, 2003). While we still do not know how the brain creates the mind, represents information, and allows us to reason, we do have consensus that the brain is made up of a large number of simple cells called *neurons*. The number of neurons we have is estimated to be anywhere between ten and a hundred billion.

Each neuron is itself a simple device that receives electrochemical inputs from other neurons, and in turn generates an output signal that flows into other neurons. A biological neuron has basically three distinct components¹⁷ as shown in Figure 4.19,

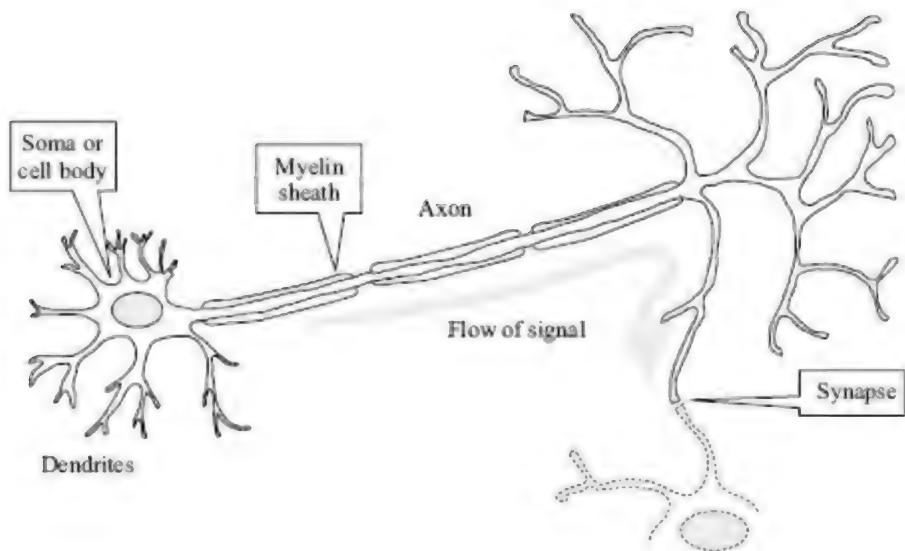


FIGURE 4.19 A biological neuron from the brain receives several inputs via its dendrites and sends a signal down its axon. The axon branches out as well and connects to dendrites of other neurons via synapses, which transmit the signal chemically to the other neurons. The shaded portion of the soma is the nucleus of the cell.

- The *soma* or the central part of the neuron is the cell body that contains the nucleus. Every neuron is surrounded by a plasma membrane containing proteins that make it behave like a gate. They hold charged particles that may be transmitted across ion channels, based on a voltage difference or sometimes on chemical properties. Neurons communicate with other neurons via electrical and chemical *synapses*.
- The *dendrites* are cellular extensions with many branches forming what is known as a *dendritic tree*. The dendrites receive signals from other neurons they are connected to via synaptic connections.
- The *axon* carries the signal that the soma generates to other neurons. The axon can be very long, often being thousands of times the diameter of the soma. It is usually coated by a *myelin* sheath that provides insulation, enabling the transmission of electrical signals over longer distances. The terminals of the axon branches have the synapses that connect to other neurons via their dendrites, or even on the soma, and transmit signals chemically across.

Each neuron continuously receives signals through its dendrites in the form of spikes, and, from the information processing point of view, at some point based on some criterion, it sends a signal down its axon which is transmitted to other neurons. Figure 4.19 shows a schematic diagram of a neuron and illustrates a synaptic connection with the dendrite of another neuron.

The central nervous system of a biological entity is the information processing infrastructure of the colony of cells, that is the body. There are different types of neurons carrying out different functions in the nervous system. Functionally, its neurons can be classified as being of three types.

- *Afferent neurons* or *sensory neurons* capture information from tissues

and organs and convey it to the centre of the nervous system, or the brain.

- *Efferent neurons* or *motor neurons* carry signals from the central nervous system to the cells that control the muscles and other systems.
- *Interneurons* are those neurons that connect to each other in the central nervous system. These are the neurons we are interested in, because they make up what we call the brain.

The behaviour of each individual neuron is relatively simple. It receives signals from multiple neurons via its dendrites and produces its own signal under certain conditions. One can model this functionality as shown in Figure 4.20, where its output y is some function of all the inputs x_1, x_2, \dots, x_n it receives.

That is,

$$y = f(x_1, x_2, \dots, x_n) \quad (4.2)$$

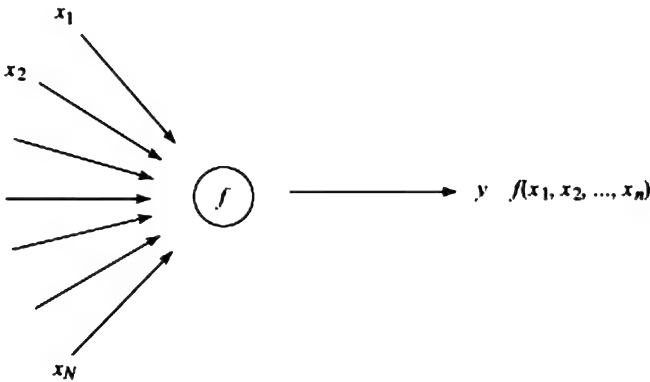


FIGURE 4.20 The neuron is a simple device that computes some function of all the inputs it receives from the other neurons.

The simplest model of the neuron applies some function to the weighted average of all the inputs. The output y_i of the i^{th} neuron is given by,

$$y_i = \phi(\sum_k w_{ki} x_k) \quad (4.3)$$

where x_k is the input received from the k^{th} neuron, and w_{ki} is the weight of the connection between the k^{th} neuron and the i^{th} neuron. In practice, the function used is

$$y_i = \phi(\sum_k w_{ki} x_k + b) \quad (4.4)$$

where b is a bias, whose role becomes clear in Chapter 18, when we take up training.

The function ϕ is a function applied to the weighted sum, and is known as an *activation function* which controls the output that the neuron generates. As we will see in Chapter 18, it is necessary that this function be a nonlinear function, if the network of neurons is to be able to do interesting

computations. The simplest function is the *Threshold function* defined as follows,

$$\begin{aligned}\phi(x) &= 1 \text{ if } x \geq 0 \\ &= 0 \text{ if } x < 0\end{aligned}\tag{4.5}$$

With this function, the neuron generates an output 1, when the input to it crosses the threshold 0 (the bias b plays a role here), and outputs 0 otherwise. This is also known as *Heaviside function*, and this model of the neuron is also known as the McCulloch-Pitts model after Warren S McCulloch and Walter Pitts who first described it (McCulloch and Pitts, 1943).

Another function used is the sigmoid function, that we have seen earlier in Section 4.2.

$$\phi(x) = 1/(1 + e^{-ax})\tag{4.6}$$

where a is a slope parameter that controls the shape of the sigmoid function in a manner similar to what is done by the parameter T in Figure 4.6. Here, the output is not a step function, but a graded increase, asymptotically reaching the value 1.

Many more refined models of the neuron have been created in an effort to model the biological neuron, keeping in mind that the output of the neuron is a temporal response in which a neuron generates output in a burst, only when certain conditions are met. That is beyond the scope of this text.

But we are concerned with the fact that colonies of interconnected neurons become the complex information processing machine, that is the brain. How does that happen? Constructing brains directly from neurons though is still beyond our means. But we do know that representing and reasoning of information is possible because of the connections between neurons. It is the topology of the connections between these billions of neurons that determines what thoughts go on in our heads. The brain is a large network of simple processing elements, a *neural network*. The brain is an *emergent system*, in which complex behaviour emerges from a system of many interacting simple elements, the neurons. We illustrate emergent systems in the following section.

If the performance of neural network is directly dependent upon the connectivity and graded influence of different neurons then how does one decide how much one neuron affects another? In other words, how does one decide the weight of each neural connection? The approach taken by the Artificial Neural Network community is learning. We shall look at it in Chapter 18. The basic idea is to develop algorithms that adjust the weights, based on training data that is shown to the system.

The approach to learning is also inspired by how human brains learn. A newborn human child does not have a very effective brain. Initially, the human brain is overly connected. But gradually, with a lot of “training” by parents and other adults, the child’s brain breaks some connections and strengthens others; in the process developing a keen and incisive brain that gives the human a definite advantage over other species.

In humans, the brain makes connections in early childhood. But we can

also reconnect—for example, visually impaired people can use that unused part of the brain for other tasks. As an example, blind people who learn to read Braille use the part of the brain that is used in vision for processing tactile information, doing it much better than the rest of us.

The human brain thus is a prime example of a complex system arising out of interaction between many simple elements.

4.6 Emergent Systems

Ever since people have begun to understand the mechanisms of life, and the interplay between semiotic and genetic interaction, there have been efforts to simulate the process of emergent complex systems by putting together the basic set of simple parts and then letting “nature do the rest”. Prominent amongst these are efforts to mimic life in the field of *artificial life*. One such effort is described (Cliff and Grand, 1999; Grand, 2001) in which the authors adopt a ‘bottom-up’ approach of putting together artificial creatures.

The fact that systems of simple elements can come together to display complex behaviour was illustrated by the British mathematician John Conway in a cellular automaton that has come to be known as Conway’s *Game of Life*, first publicized by Martin Gardner (Gardner, 1970). The *Game of Life* is played on an infinite two-dimensional grid of cells, each of which can be in one of two states, alive or dead. The “health” of a cell is directly influenced by the eight cells surrounding it. The automaton has simple rules for life and death:

1. Cells that are alive will continue to live if they have exactly two or three living cells in their neighbourhood. Otherwise, they die of loneliness (less than two neighbours around) or overcrowding (more than three neighbours alive).
2. Cells that are dead are born again if they are surrounded by exactly three living cells.

These rules are applied *simultaneously* to the entire grid, generating a new population from an old one. The system is a *cellular automaton*. The rules are simple and the reader is encouraged to implement the game (or download one of the numerous implementations available on the Web). The game became extremely popular because of the surprising patterns that evolve from these simple rules. It has a number of well documented patterns which emerge from particular starting positions. Soon after publication, several interesting patterns were discovered, including the ever-evolving *R-pentomino* (more commonly known as “*F-pentomino*” outside the game), the self-propelling “glider”, and various “guns” which generate an endless stream of new patterns¹⁸, all of which led to an increased interest in the game. Figure 4.21 below shows the well known Glider. The pattern on the left goes through four transformations that result in the same pattern, but *shifted* one step right and down in the grid. Over a period of time, it appears that the pattern is moving across the screen.



FIGURE 4.21 The well known Glider in the Game of Life. By the time the above shape has gone through the transformations, it forms the same pattern shifted one step down and right. Over a period of time, it looks like the pattern is shuffling or gliding across the screen.

This “illusion” of movement is interesting because what is happening is that different cells are dying and coming alive. In that sense, the moving pattern is more like a wave in which the carrier does not move, but the signal¹⁹ does. Grand (2001) gives another interesting example of an illusion of being. Some types of clouds that seem to hang still over a mountain range, are in fact physically moving moisture particles that condense and become visible only when passing over the mountain range, and then evaporate again, giving us the illusion of a stationary (orographic) cloud.

Karl Sims demonstrated that one could “put together” working designs by a genetic system that experimented by putting together parts and testing them for their functionality in a simulated, three-dimensional physical world (Sims, 1994). The physical-world simulation included articulated body dynamics, collision detection, collision response, friction, and an optional viscous fluid effect. In this simulated world, he put in creatures evolved for specific tasks or behaviour like swimming, walking, jumping and following other creatures. Different fitness functions are used for each of the above tasks. He developed a genetic representation that uses nodes and connections to describe both the morphology (the form and shape of the creature), as well as the neural circuitry used to control the system. The genotype of the system is a directed graph. The evolution of creatures begins by first creating a population of these genotypes. This could be done by creating random graphs, or by some other means like handcrafting them, or carrying some forward/across different runs of the system. The phenotype of the creatures is made up of a set of three-dimensional rigid parts, connected by flexible joints, powered by muscles. Two examples of simple genotypes and the corresponding phenotypes are shown in Figure 4.22 below, taken from his paper.

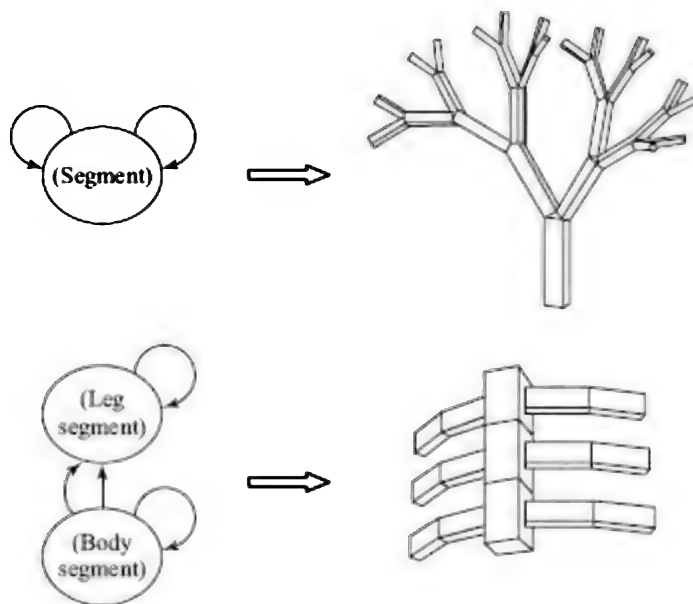


FIGURE 4.22 Two recurrent genotypes and their corresponding phenotypes. The genotype is a directed graph, and the phenotype a hierarchy of 3-D parts. Figure from (Sims, 1994).

Each graph contains the development instructions for growing the creature. Each node in the graph contains information describing a rigid part. The shape and size is described by a set of dimensions, and a joint type describes how it is connected to other parts. Each connection also contains information about the placement of the connected part in terms of position, orientation, scale and reflection. The artificial creature is controlled by a virtual brain, made up of a collection of neurons, distributed across the body parts.²⁰

The brain is a dynamical system that accepts input sensor values, and generates output to control the system. The output values are applied as forces or torques at the degrees of freedom of the body's joints. The sensors can sense the position of body parts contact with body parts and light. The brain has internal neurons that define its internal state and play a role in determining the 'out' signals. Thus, the creature is more than a reactive system. While neurons are distributed along body parts, there is also one set of "central" neurons that allows for global synchronization of movement and central control.

When a creature is synthesized from its genetic description, the neural components described within each part are generated along with the morphological structure. This causes blocks of neural-control circuitry to be replicated, along with each instantiated part, so that each duplicated segment of a creature can have a similar but an independent, local control system. These local control systems can be connected to enable the possibility of co-ordinated control.

Figure 4.23 below from (Sims, 1994) shows some of the virtual creatures evolved for walking. The author says that "*The walking fitness measure also produced a surprising number of simple creatures that could shuffle or hobble along at fairly high speeds. Some walk with lizardlike gaits using the corners*

of their parts. Some simply wag an appendage in the air to rock back and forth in just the right manner to move forward. A number of more complex creatures emerged that push or pull themselves along, inchworm style. Others use one or more leglike appendages to successfully crawl or walk. Some hopping creatures even emerged that raise and lower armlike structures to bound along at fairly high speeds.”

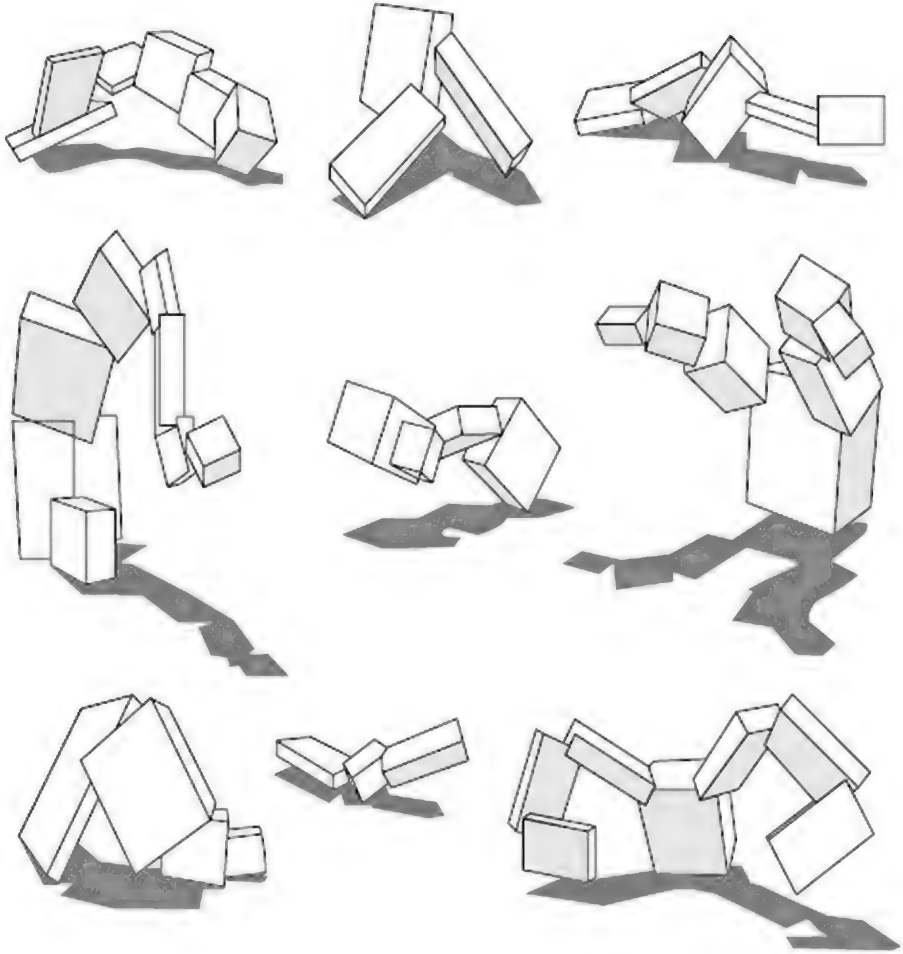


FIGURE 4.23 Different designs evolved for walking creatures. Figure from (Sims, 1994).

The central theme in evolutionary or emergent systems is to provide the basic building blocks needed for the full system, and then letting some kind of recombination and selection process work on a population. While many interesting systems have been produced, the building blocks for most have to be carefully engineered. Nature, on the other hand, has used this process extensively from scratch. We can even say that Nature *is* this process of assimilating complex systems through *natural selection* from a population of designs.

Nature has been so successful at it that humans have wondered at the

world around us and concluded that it must be the work of a master craftsman, a super being. Beginning with the work of Charles Darwin, only in the recent past have arguments for the natural evolution of our world been accepted (see (Dawkins, 1986, 1996)). Initially, the rate of change must have been very slow. To quote (Hoffmeyer, 1994a) “... *evolution spent two billion years to create that enormously complicated web of chemical habits which we call the eukaryotic cell, and the genome would not work if it were not put into the context of the historically appropriated competence of that cell. Ecosystems would not be stable were it not for the millions of semiotic processes built on habits which themselves were formerly built on other habits. And the human brain would not function without the historically developed patterns of communication between many billions of highly organized nerve cells.*” Only after it had stumbled upon the building blocks of life as we know it, did the natural process accelerate, eventually leading to an explosion of myriad life forms. The basic building blocks may be relatively small, but the combination of the genetic material has survived in many many different phenotypes.

Genetic or evolutionary algorithms operate by recombining and transmitting information from individuals in one generation of the population to the next. Many life forms increase the chances of their survival by developing mechanisms for transmitting information amongst individuals of the same generation, creating a semiotic system. We look at a population based optimization approach, based on sharing information below.

4.7 Ant Colony Optimization

Ants have received much attention due to their ability to act in a coordinated manner. A *colony of ants* is able to produce complex problem solving behaviour through semiotic interaction (as described in the boxes in this chapter). In particular, scientists have been impressed by their ability to find shortest paths between a food source and their nest, and being able to quickly discover new optimal paths when the world changes in some way (Goss, 1989; Beckers, 1992).

This complex adaptive behaviour is achieved by a *simple mechanism* of leaving *pheromone* trails and following these trails. Successful behaviour is *reinforced* by the phenomenon of more ants following the trail and returning successfully (and quickly), in the process *depositing more pheromone* on the trail. Ants that wander off along other paths will not have this continued strengthening of their trail, and will not attract other ants.

Consider a thought experiment in which two ants go off in two directions in search of food. Both leave trails behind them, and the ants that come out of the nest after them randomly follow one of the trails. If one of the trails leads to food then the ant will bring the food back along that trail, adding more pheromone on the way back. More ants will follow, and return with more food, making the trail even stronger. Combined with the fact that the trails have a natural tendency to dry up (after all they are made up of minute chemical deposits), the overall effect will be that the food bearing trail will eventually become strong enough to attract all the emerging ants, and further activity will be focused entirely along the successful trail.

Furthermore, if the world changes, say an obstacle is suddenly placed in their paths (see Figure 4.24 below), the ants will not be able to follow the trail, but will have to “search” again. Both the outgoing ants and the returning ants will come up against the obstacle, and will have to choose between a left and a right turn. Some will choose left and some right, and there will be more choices for them later, but eventually the ants making the correct choices will rejoin the trail sooner. This will reconnect the other ants, and this new trail will quickly be adopted as the trodden path.

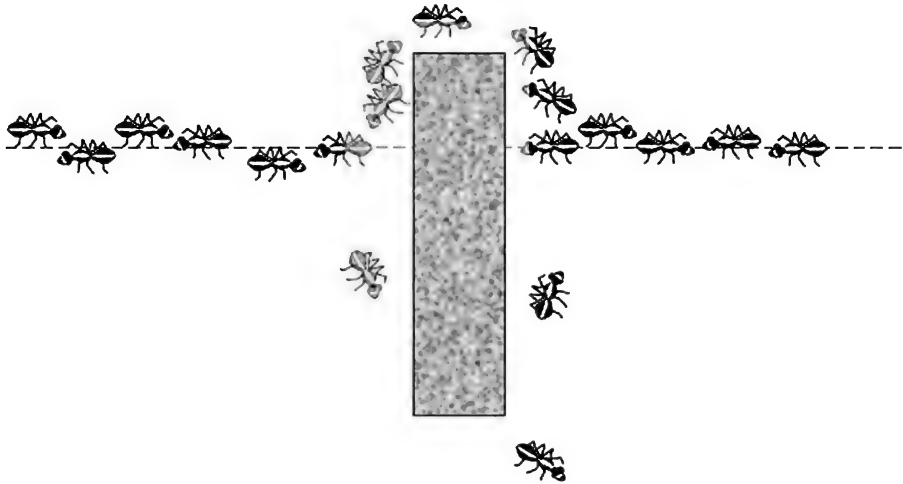


FIGURE 4.24 When an obstacle is placed on the pheromone trail of an ant colony, they quickly find the shortest diversion around it.

In the absence of a trail, the real ant presumably does a random walk. In fact, the ants are known to follow the trails only probabilistically. The stronger the pheromone trail, the more likely that an ant will follow it. In the computational version of the algorithm, (see (Dorigo, 2004)), Ant Colony Optimization (ACO) combines this pheromone pull with the innate problem solving strategy of an agent. ACO thus refers to a set of agents working together in such a way that the agents have a tendency of following and reinforcing the “trails” of other agents, in addition to their own strategy. Given a choice of moves, the agent (an ant) will be influenced by two things. One is its own heuristic information that indicates a preference from the available moves. The second is the amount of pheromone left by other ants.

Pheromone trails are simple semiotic means of sharing experience. The process works when the pheromone deposits reinforce good moves as opposed to bad ones. In the case of food finding activity, the ants travel at a constant speed and leave a constant amount of pheromone that evaporates at a constant rate. On the paths that lead to food, the ants will return quickly, adding more pheromone, and thus strengthening the shortest path to the food. In the example below, we look at the *TSP* problem that was used originally by Dorigo et al. (Colomni, 1991). In the ACO algorithm for *TSP*, a population of ants are used to construct a set of tours. *After* the tours have been constructed, each ant deposits an amount of pheromone on the edges, making up the tour that is *inversely* proportional to the cost of the complete

tour. In this way, the ants finding shorter tours will leave stronger pheromone trails, which will then attract more ants on those segments.

Let there be n cities in the *TSP* problem. Let $\tau_{ij}(t)$ represent the amount of pheromone on segment from the i^{th} city to the j^{th} city at time t , which is the start of a new tour construction. Then after n moves, each ant would have constructed a new tour and the pheromone trails will be updated as follows,

$$\tau_{ij}(t+n) = (1 - \rho) \tau_{ij}(t) + \Delta \tau_{ij}(t, t+n) \quad (4.7)$$

where $0 \leq \rho \leq 1$ is the coefficient of pheromone evaporation. By choosing an appropriate value of ρ , one can control how long the effect of pheromone deposit will last. The second term in the above equation is the sum of the total pheromone deposited by all the ants on the segment from the i^{th} city to the j^{th} city during this cycle, computed as follows, for m ants.

$$\Delta \tau_{ij}(t, t+n) = \sum_{k=1}^m \Delta \tau_{ij}^k(t, t+n) \quad (4.8)$$

The amount of pheromone deposited by the k^{th} ant is given by,

$$\Delta \tau_{ij}^k(t, t+n) = \begin{cases} Q/L_k & \text{if the } k^{\text{th}} \text{ ant travels on the segment } i-j \\ 0 & \text{otherwise} \end{cases} \quad (4.9)$$

where Q is a constant, and L_k is the cost of the tour found by the k^{th} ant. Initially $\tau_{ij}(0)$ is set to a small value Δ for all segments.

The basic outline of the ACO algorithm for *TSP* is described in Figure 4.25.

```

TSPACO()
1  Initialize  $\tau_{ij}(0) = \Delta$  for all segments  $i-j$  in the problem
2  repeat
3      Construct the tour for each of the  $m$  ants
4      Remember the best tour when a better one is found
5      Update the pheromone levels for each segment  $\tau_{ij}(t+n)$ 
6  until some termination criteria
7  return the best tour

```

FIGURE 4.25 In the ACO algorithm for TSP, a set of m ants construct tours by a greedy algorithm that moves probabilistically to the next city. After the tour construction, each ant deposits pheromone inversely proportional to the cost of its tour. This process is repeated a number of times, retaining the best tour found.

Each ant constructs a tour in a greedy manner. It starts at some city, and at each stage moves from the i^{th} city to an allowed neighbour j with a certain probability governed by the amount of pheromone $\tau_{ij}(t)$ on the $i-j$ segment, and η_{ij} its heuristic estimate of how good the segment $i-j$ is. The probability of the k^{th} ant moving from the i^{th} city to the j^{th} city is given by,

$$P_{ij}^k(t) = \begin{cases} \frac{[\tau_{ij}(t)^\alpha + [\eta_{ij}]^\beta]}{\sum_{h \in \text{allowed}_k(t)} ([\tau_{ih}(t)]^\alpha + [\eta_{ih}]^\beta)} & \text{if } j \in \text{allowed}_k(t) \\ 0 & \text{otherwise} \end{cases} \quad (4.10)$$

where $\text{allowed}_k(t) = \{j \mid j \notin \text{tabu}_k(t)\}$, and $\eta_{ij} = 1 / \text{distance}(\text{city}_i, \text{city}_j)$ also called the visibility of city_j from city_i . The algorithm maintains a set $\text{tabu}_k(t)$ for each ant k , that contains the cities already visited by the ant in that cycle of tour construction.

In the *TSP-ACO* algorithm, the amount of pheromone on a segment gives an indication of the goodness of a move, based on the accumulated experience of the population. To achieve this, the amount of pheromone deposited by each ant is directly determined by the goodness ($1/\text{cost}$) of the solution found. Real ants move at a constant speed leaving behind a constant amount of solution. The ant colony manages to reinforce good solutions, simply because of the fact that ants using shorter routes to food come back faster, hence replenishing the pheromone levels faster.

The ACO algorithm then can be seen as a parallel, randomized search algorithm that converges towards good solutions by a process of learning in which, each agent communicates some information about the goodness of solution components to a common pool. As more and more agents explore the good components, more agents use them to build solutions. In some sense, this behaviour is similar to the one in GA, except that there it is the solutions themselves that make up the populations. In ACO, the solutions are not coded, but simple agents repeatedly construct solutions, exploiting the information (or experience) of earlier attempts by the entire population.

The ACO algorithm also has similarities with the *Simulated Annealing* algorithm. In both, a move is made probabilistically to a neighbour. SA works with a single solution, perturbing the solution probabilistically. The SA algorithm first chooses a neighbour and then decides whether to move to it or not. ACO looks at all neighbours to pick one to move to. In SA, the decision is based only on the goodness of the move determined by ΔE (temperature T is an external parameter). The decision in ACO is based on a combination of memetic signals of experience and the estimated goodness as determined by the heuristic function.

Real ant colonies survive because the simple interaction between simple agents has clicked (or emerged) as a good food finding behaviour. An individual ant, viewed as a system itself, is quite simple. The apparent complexity of its behaviour over time is largely a reflection of the complexity of the environment (terrain) in which it finds itself. See (Minsky, 1986; 2006) for more on emergent behaviour from simple elements in human brains. The strategy of ants is hardwired for food finding. It has survived so long because food for energy is a primary need for life, and ants probably don't need much else. Observe that the strategy is good for ant colonies as a whole, or ant species in general. Consequently, it is also good for most individual ants; but it may not necessarily be good for a given individual ant. Humans on the other hand are much more complex creatures. They have evolved brains, which they consciously use to work *for the survival of the owner* of the brain, the

individual²¹. In the process, we have evolved many secondary goals such as old age security, self actualization goals, recreation and entertainment goals. We pose and solve many different problems in many different ways. Communication between humans has gone much beyond, leaving simple chemical trails. We are able to model our environments, reason with our models, and communicate our thoughts and reasons to others via spoken and written language. The average human receives a considerable amount of information from society through her lifetime. Our knowledge and language systems need a much more detailed study that goes beyond memetic habits. We will look at the representation and use of knowledge in later chapters.

4.8 Discussion

Search spaces for most problems are too large to be searched exhaustively. Complete algorithms like *Best First Search* may run out of memory space for large problems. Local search algorithms like *Hill Climbing* require small, usually constant, amount of space. While they do mitigate the space requirements, *Hill Climbing* ends up with another problem. They may not find the solution. *Hill Climbing* is a simple greedy algorithm that puts its entire faith into the heuristic function, choosing at each stage the move along the steepest gradient as indicated by the heuristic function, and possibly getting stuck on a local optimum with no way of knowing whether a better one exists. The *HC* algorithm *exploits* the heuristic function single mindedly. A *Random Walk* on the other extreme completely ignores the heuristic function, choosing instead to *explore* the search space. One redeeming feature of a random move is that it cheerfully goes past a local optimum, and this is what most methods that mix exploitation and exploration want to capitalize upon.

Simulated Annealing does this by probabilistically allowing random moves. At any point of time, it looks at one neighbour, and may move to it with a certain probability. The probability depends upon the amount of exploitation the move does. The better the neighbour is, the more likely the move. It can also move to a worse neighbour, though with a small probability. Even if no good moves are available at a node, it will eventually make a bad move too, and get off local optima. The probability also depends upon what stage the search is in. In the initial stages, *SA* is biased towards exploration, and the probability is high, irrespective of the value gain or loss, but it gradually settles down for exploitation. Towards the end, the chances of it being in the vicinity of the solution is higher, and hopefully the heuristic function will then do the job, and guide it to the solution.

Genetic Algorithms aim to mimic the way nature does design. The genetic makeup of each creature in the real (and artificial) world determines how it will grow, and act in the world. Viewing each creature as a program written with genes, nature has developed a mechanism of allowing creatures to mate and mix genes. The ones who get to do so are the strongest, in terms of survival abilities²². Nature allows the best (as determined by survival) designs to reproduce more. The process of reproduction mixes genes of the two parents. This may produce an offspring better than both, if it combines successful features from both. *GAs* attempt to mimic this selection process by allowing

components of solutions to be inherited as a mix from two parents, and selecting the good solutions based on an explicit, fitness function. The general idea is that such churning of “genes” in a population of solutions will result in fitter populations, just as it has resulted in fitter species in nature. This happens when random mixing of genes from fit parents produce a fitter offspring. In this case, the offspring will get more chances to reproduce and pass on its genes. The system improves from generation to generation, passing on information (of the solution design) encoded in the genetic make-up.

None of the methods for finding optimal solutions that we have seen are foolproof. None work in all situations. One can often try combinations of algorithms to develop hybrid algorithms. Some of the ideas of combining algorithms are as follows. One could start with a fit population for *Genetic Algorithm* by producing solutions by other methods like the *Iterated Hill Climbing*. Conversely, one could apply *Genetic Algorithms* to a population, and then follow up with *Hill Climbing* or *Simulated Annealing* on each member. Or, one could find solutions from ACO and improve upon them by other methods.

Ant Colony Optimization methods aim to mimic how, in a given population, individuals can cooperate by sharing information as a matter of habit. Ants, by themselves, have a limited view of the world. In their search for food, the ant has a tendency to follow a trail left by other ants, and in turn leave a trail for others to follow. Ants that find food will quickly retrace their steps, strengthening the trail; and this simple process reinforces the food finding trails, leading to a steady procession of ants towards the food source.

Ants are in fact doing a simple form of experience sharing. This is hardwired into their genetic make-up. They do not conceptualize their surroundings. They do not represent knowledge and facts, and reason with them. They are simply programmed to leave and follow trails. Consequently, their experience sharing behaviour is not flexible, and their lifestyles are simple. Humans on the other hand conceptualize their surroundings; have developed the ability to represent things, to reason about consequences, and language to share knowledge with. Consequently, our experience sharing behaviour is much more diverse. We can specialise in many areas and acquire experience in them. We can articulate this experience in fables, facts, stories, and text books to pass them on to others. The human species has accumulated knowledge over generations. We pass on to our offspring not only our genes, but also knowledge in various forms. This ability to build up experience in a cumulative manner has resulted in the human species dominating all others. Our knowledge of our surroundings is evolved to such an extent that we are even able to contemplate the harmful effects human civilization is having on the environment and ecology in the longer timeframe, so that the actions that will avoid a catastrophe (and destruction of the species) are still on the *OPEN* list of humankind.

In our quest for randomized search algorithms, we also ran into biological systems that can be seen as employing search. We also learnt that such randomized methods often work well when we operate with populations of candidates. This led us to emergent systems, where populations of simple elements can be seen as a more complex system. Human brains and ant

colonies are examples of such complex systems. At a higher level a research institute, a manufacturing company, or an academic department can be seen to be more complex than the individuals that make it up. Music lovers would also recognize the fact that a musical note by itself is simple in nature, but it is a combination of notes that makes up a melody or a symphony.

The point we make in this chapter is that hard problems cannot be solved by brute force. One needs either to have good heuristic functions to guide search, or one needs to share and exploit experience. Either way, the tool to cut through the combinatorial explosion has to do with knowledge. In later chapters, we will look at knowledge and experience in more detail.

Meanwhile, in the next chapter we turn our attention to those approaches to problems that can guarantee optimality. We explore the use of heuristic knowledge to search for optimal solutions, and push the envelope of the size of problems that can be solved in a reasonable time with small space requirements.

Further Reading

Many of the topics covered in this chapter are research areas in their own right. Optimization is an active field of work, and stochastic local search methods have been studied in great detail. A comprehensive book on the topic is the book by Hoos and Stutzle (2005). An interesting account of problem solving methods is given in the book by Michalewicz and Fogel (2004). Goldberg has two books (1989, 2002) on genetic algorithms and innovation in design. Dorigo has a book on ant colony optimization (2004). There are several books on artificial neural networks. A popular one is the book by Hassoun (1996), and another by Yegnanarayana (1999).



Exercises

1. What do the terms ‘exploitation’ and ‘exploration’ refer to in the context of search?
2. Describe the *Simulated Annealing* algorithm. When does one prefer to use the algorithm? What is the role of the parameter “temperature” in the algorithm?
3. When does *Simulated Annealing* perform better than *Hill Climbing*? How is this better performance achieved? Would you ever prefer *HC* to *SA*? If yes, when?
4. Compare the performance of *Iterated Hill Climbing*, *Simulated Annealing* and *Genetic Algorithm* on the instance of *Uniform Random k-SAT* problems of Exercise 3.10.
5. What is the motivation and strategy behind Genetic Algorithms? Under what conditions are GAs likely to perform better than other optimization algorithms?
6. Experiment with different population sizes for the Genetic Algorithm for solving the SAT problem. Generate different *Uniform Random k-SAT* problems of different sizes. For each problem, generate different, random

candidate solutions and apply *Hill Climbing* to each candidate before adding it to the population. Does the performance improve?

7. In Section 4.4.3, we have observed that not every permutation of numbers representing cities in the adjacency representation is a valid tour. Since every permutation in the path representation is a valid tour, does it mean that there are some tours that are not represented in the adjacency representation?
8. Given the *TSP* candidate solution (DHGBFCAE) in path representation, remove three edges and show the neighbours generated by a 3-edge exchange operator. How many neighbours in all does this operator generate for this candidate?
9. Given the two tours in the *TSP* problem, (FICDEBAHG) and (EAGIDBHFC), illustrate,
 - (a) The order crossover
 - (b) The partially mapped crossover
 - (c) The cycle crossover
10. Create Ordinal Representations for the two candidates in the previous question.
11. Given the cost matrix in the accompanying table, determine the estimated cost of a *TSP* tour that contains the segment DE, but excludes the segment AC. Describe your estimating function. What properties should the estimating function have to devise a good *TSP* solver?

	A	B	C	D	E
A	0	70	20	50	60
B	70	0	40	10	30
C	20	40	0	30	40
D	50	10	30	0	70
E	60	30	40	70	0

12. A *TSP* problem with N cities is being solved by the ant colony optimization problem using M ants. An edge E_{ij} has $\sigma(t)$ amount of pheromone at time t . When will the amount of pheromone on this edge change and by what amount? Be precise.
13. Explore the World Wide Web to get some facts about the human brain. How many neurons do we have? What is the size of the soma? How long can the axon be? How many neurons is a given neuron connected to? Does the number of connections increase or decrease with age?
14. Explore the books Godel Escher Bach (Hofstadter, 1979) and The Mind's I (Hofstadter and Dennett, 1981). Read the chapter on 'Ant Fugue', and study how a colony of ants can be treated as a more complex organism.

¹ If in the candidate solution, each city was connected to its two nearest neighbours, then the solution will clearly be optimal, but not all optimal solutions satisfy this property.

² The name *annealing* comes from physical processes in which materials are cooled at a controlled rate with the aim of stabilising in minimum energy states. Since this can be seen as a problem of minimization, the

term 'energy' has been carried to the evaluation function value in the computational version of this physical process.

³ http://en.wikipedia.org/wiki/Cope's_rule

⁴ <http://en.wikipedia.org/wiki/DNA>

⁵ The strategy adopted by human beings involves representation and modelling of their environment, including themselves, so that it is possible for human beings to contemplate and reflect upon their own strategy, fine tune it to the world, and even create machines and exploit other life forms to their advantage. Interestingly, human beings can contemplate human beings contemplating human beings contemplating contemplating their strategy, ending up in all kinds of philosophical conundrums about reality and what is really happening out there.

⁶ One supposes that there is no fundamental reason that pairs are used, except that this process is the one that has persisted. In principle, the offspring could be a result of a recombination of genes from more than two parents, but perhaps nature did not find a mechanism to do so. Observe, that handling such pairs is also commonplace in computer science. We represent numbers in binary form, we use two valued logics of true and false sentences, and most sorting algorithms compare two elements at a time.

⁷ The cheetah is an example of such an event happening. The animal was so well suited to its predator life form, that the species stabilised into low genetic variation. With human civilization changing the face of the world totally, the same cheetah is a misfit and on the verge of extinction.

⁸ <http://www.freemars.org/jeff/2exp100/answer.htm>

⁹ *Lisp* has a built-in feature to handle large numbers. It has long been a favourite language of AI researchers, primarily because it allows dynamic structures to be built naturally, and its functional style allows the creation of domain specific operators easily. It has been on a little bit of a decline with the advent of other object-oriented languages, and a diminishing community makes it daunting for new entrants to try their hand at it.

¹⁰ <http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/>

¹¹ Quoted from Wikipedia:
http://en.wikipedia.org/wiki/Traveling_salesman_problem

¹² The triangle inequality states that the sum of the distances along two sides of a triangle is larger than the distance along the third side.

¹³ The development of vertebrates (creatures with backbones) in nature is an example. The set of genes for developing the vertebra can be thought of as a BB. However nature took quite long to arrive at designs using vertebras, with the first appearance around the Cambrian explosion about 530 million years ago, much later in the journey of life begun 3.5 billion years ago.

¹⁴ Notice that we are not saying that they *will* come together to display complex behaviour. They might and when they do it is up to the selection mechanism to allow them to persist.

¹⁵ <http://www.molbio.ku.dk/MolBioPages/abk/PersonalPages/Jesper/Hoffmeyer.html>

¹⁶ http://en.wikipedia.org/wiki/Common_Era: The Common Era (CE),

sometimes known as the Current Era or as the Christian Era, is the period of measured time beginning with the year 1 on the Gregorian calendar. The notations CE and BCE (Before the Common Era or Before the Christian Era) are alternative notations for AD (*Anno Domini*, Latin for “in the year of the Lord”) and BC (Before Christ), respectively. The CE/BCE system of notation is completely chronologically equivalent with dates in the AD/BC system, i.e. no change in numbering is used and neither includes a year zero. The abbreviations may also be written CE and BCE.

¹⁷ See <http://en.wikipedia.org/wiki/Neurons>

¹⁸ See Wikipedia for a set of illustrative patterns.
http://en.wikipedia.org/wiki/Conway's_Game_of_Life

¹⁹ We can extrapolate this concept to our own selves. The body we had as a baby was made up of an almost entirely different set of atoms and molecules as compared to our current body. Yet, we as selves, have existed continuously without being affected by this physical change. One recalls the anecdote about an axe belonging to Abraham Lincoln being kept in a museum in Illinois. Docents at the museum will, with scrupulous honesty, explain that over its life, the axe has had five new handles and two new heads changed.

²⁰ A similar view on the human brain is emerging nowadays. Hoffmeyer (1994) talks of a “floating brain”, functionally integrated into the body. Other people have observed the ability of our body parts to remember, for example our fingers to remember to type a password correctly, even when we cannot mentally recall it.

²¹ In fact, this is the basis of the market economy.

²² Evolutionary psychologists believe that our notions of beauty and sexual attractiveness are rooted in the judgment of ‘survivability’ of the offspring. Thus, people tend to be attracted to potential mates who will have healthy offspring.

Finding Optimal Paths

Chapter 5

Heuristic functions exploit domain knowledge to orient the search process towards the desired goal. The objective of using a heuristic function is efficiency of the *solution finding process*.

The solution that a search process returns may itself have a property or quality that may be of interest. That is the cost of *executing* the solution. This cost is something that we may want to minimize¹. For example, in transportation to, say Mars, each trip will have an associated cost, which is quite significant. Or the different operations that go into acquiring components and assembling a product would each have an associated cost. Optimal solutions would be needed, especially if the solution found has to be executed many times. In such a case, the primary concern would be solution quality, and the speed with which the algorithm finds the solution could well be secondary.

While the number of moves that make up the solution is often a good measure of cost, it may become inappropriate when different moves have different costs associated with them. For example, you may have to change a couple of buses to reach a destination. This would count as two or three moves. Hiring a cab may accomplish the journey in one move, but is likely to be more expensive. The optimization community refers to the property that we seek to minimize as the *objective function*. In general, optimization can be a computationally hard problem. In Chapter 4, we looked at some methods for optimization that were designed to find good solutions within a reasonable time. In this chapter, we investigate methods that guarantee an optimal or a least cost solution.

5.1 Brute Force

The following algorithm is guaranteed to return the optimal solution

```
BritishMuseumProcedure()  
1 Explore the entire search space  
2 return the optimal solution
```

FIGURE 5.1 The British Museum Procedure.

The algorithm is conceptually simple. Computationally, it is mindlessly expensive. We owe the name to P H Winston who asserts that the only

way to locate something in the British Museum is to explore the entire museum (Winston, 1992). The above algorithm simply searches the entire search space.

Our focus in this chapter will be to search as little of the space as possible, while guaranteeing the optimal solution.

5.2 Branch & Bound

The intuition behind *Branch & Bound* (B&B) is as follows.

- Organize a search space that does not preclude any solution. This could be the state space, in which a partial sequence of moves is extended in each move. This could also be a refinement space in which an abstract solution is refined.
- Continue looking for a solution by refining the cheapest candidate until
 - A complete solution is found
 - No candidate solution, partial or complete, with an estimated cost smaller than that of the complete solution exists

A high level algorithm is given in Figure 5.2. Here the task is to find the lowest cost path from a start node to a goal node, and the algorithm extends the cheapest cost partial solution at each stage.

```

B&B Procedure()
1  open ← {(start, NIL, Cost(start))}
2  closed ← {}
3  while TRUE
4    do if Empty(open)
5      then return FAILURE
6    Pick cheapest node n from open
7    if GoalTest(Head(n))
8      then return ReconstructPath(n)
9    else children ← MoveGen(Head(n))
10     for each m ∈ children
11       do Cost(m) ← Cost(n) + K(m, n)
12       Add (m, Head(n), Cost(m)) to open

```

FIGURE 5.2 Branch and Bound extends the cheapest solution till the cheapest solution reaches the goal.

The basic idea behind *B&B* is to *ignore* those regions of a search space that are *known* not to contain a better solution. We look at an example in which the cost of a move corresponds to the length of an edge in a graph. Then the least cost solution corresponds to the shortest path in the graph. Let the graph in Figure 5.3 represent a tiny search space to illustrate the algorithm.

B&B begins with the start node *S*. The partial cost of *S* is zero. It expands *S*, generating partial paths *S-A* with cost 3, *S-B* with cost 4 and

S-C with cost 8. These paths are stored in the list *OPEN*. S is transferred to list *CLOSED*, shown shaded in Figure 5.4.

B&B continues extending the cheapest partial path. It terminates when the goal node is picked for expansion.

The example in Figure 5.4 shows *Branch & Bound* extending partial solutions. This is an example of state space search. We have seen earlier (Chapter 3) that when the search space is made up of candidate solutions, we search in the solution space.

If the candidate solution is only partially specified then we can think of it as a set of solutions that share the specified part. A refinement operator partitions this set into two sets by specifying another component of the solution. Each (complete) candidate from the set is some complete refinement of the partially specified solution. Search, then involves, decisions amongst the different possible refinements of a given partial solution (Kambhampati, 1997). That is, search involves choosing a refinement of some candidate by specifying more information. It is interesting to note that the state space search algorithms seen earlier are a special case of refinement, in which the partial solution specifies the path from the start node to some node *n*, and the choice is between different extensions of the partial path.

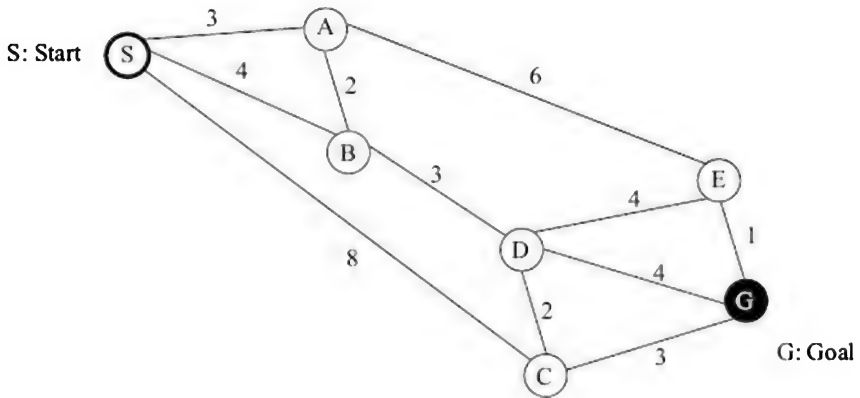
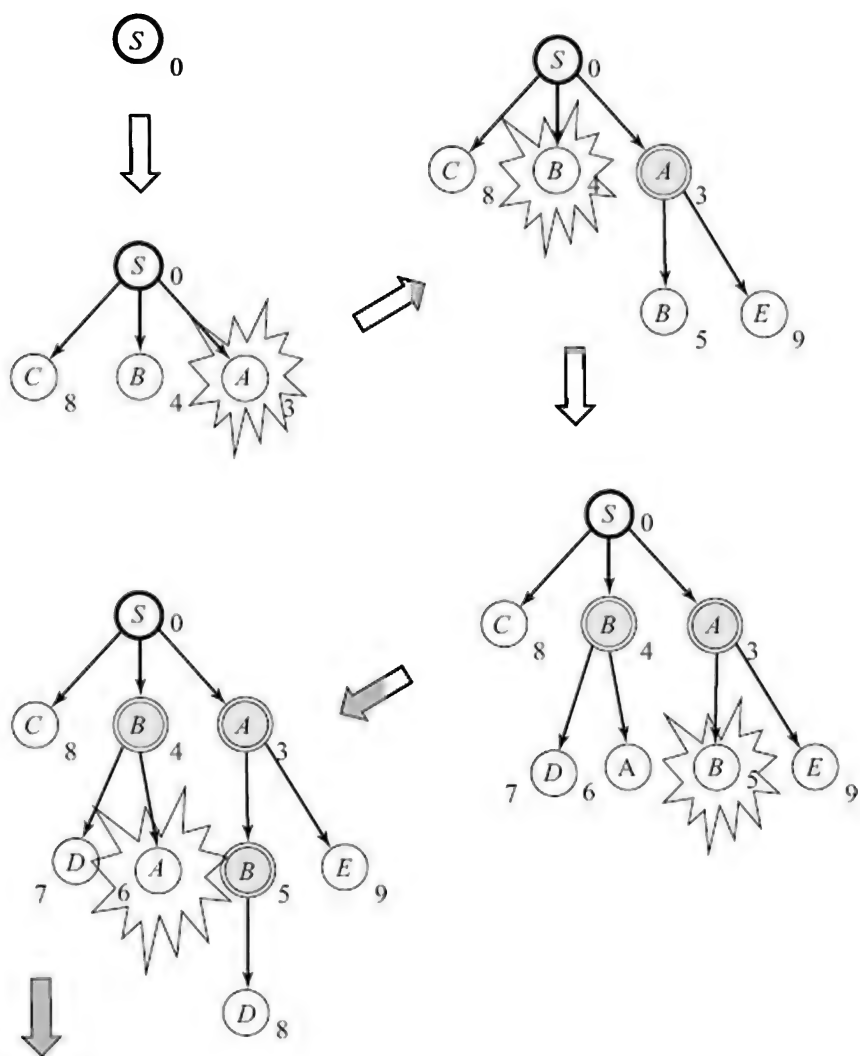
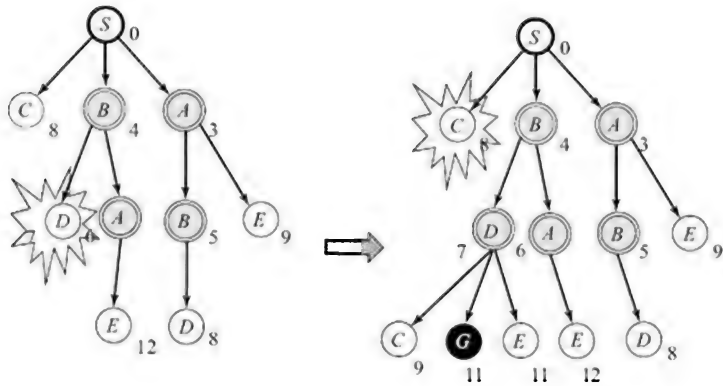
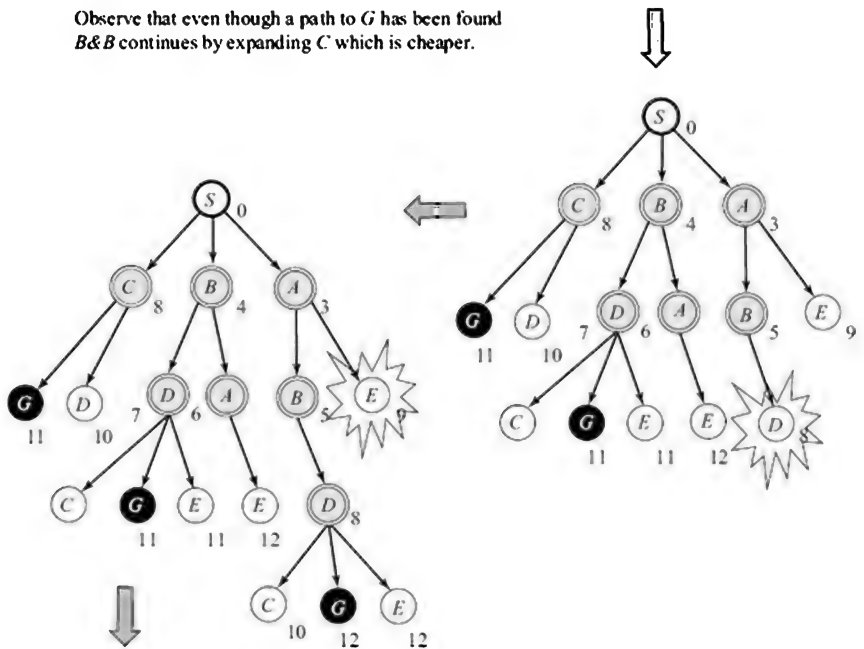


FIGURE 5.3 A tiny search graph.





Observe that even though a path to G has been found $B\&B$ continues by expanding C which is cheaper.



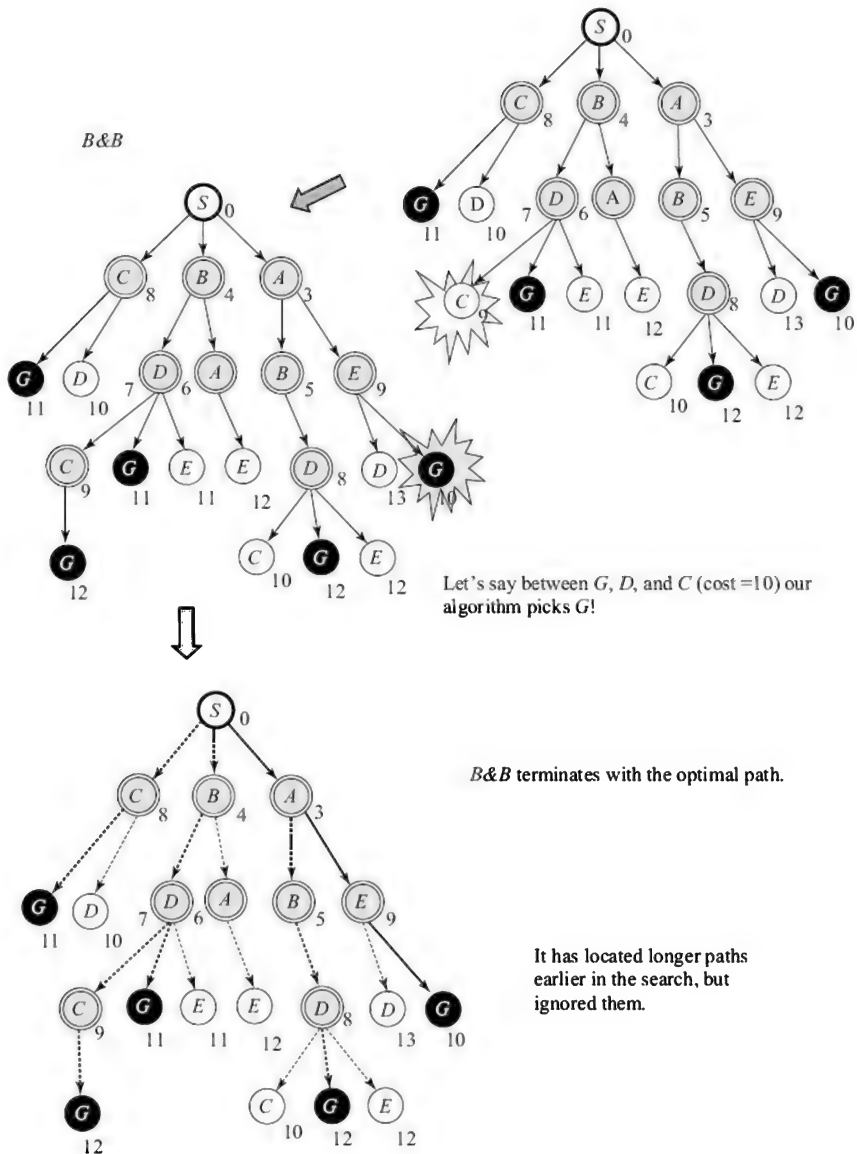


Figure 5.4 Branch and Bound explores the tiny search graph.

We consider the travelling salesman problem (TSP) again to observe the *Branch & Bound* algorithm in a refinement space.

5.3 Refinement Search

Refinement search begins with the *set* of all candidate solutions, represented by the root node. Each new node in the search tree

represents a subset of the candidates. The node can be seen as a partial solution, in which a part of the solution is specified. In each round, *B&B* selects and refines one node representing a partial solution, by specifying some more detail. This results in partitioning the set of solutions in that node. The candidate selected for refinement is the one that appears to have the lowest overall cost.

Assuming that there is a method for estimating the cost of a given (partial) solution, *B&B* refines the solution that has the least *estimated* cost. The process of refinement continues until we have a complete solution at hand, and when no other candidate (partial) solution has a lower estimated cost. We need to ensure that the cost estimates are such that the algorithm guarantees an optimal solution. Ensuring that the estimated cost is a lower bound on the actual cost can accomplish this. That is, a (complete) solution will never be cheaper than it is estimated to be. Then, candidates with estimates higher than that of some fully refined solution can be safely ignored. It is also desirable that the estimate be as high and as close to the actual cost as possible, because that will mean faster pruning of expensive candidates. We will look at this requirement more formally, later in this chapter.

How does one get a lower bounding estimate of a partial solution? Consider a refinement space search to solve the TSP. Let the candidate solutions be permutations of the list of city names. The initial solution includes all permutations, when nothing is specified. We refine this solution by specifying specific segments in the tour. In the example below, we consider the TSP problem for five cities {Chennai, Goa, Mumbai, Delhi, Bangalore}. How do we get a lower bounding estimate for the candidate tours? Consider first the absolute lower bound for all tours, represented by the root node in the search space. We look at the cost matrix with costs in kilometres.

Table 5.1 Distance matrix for five cities

	<i>Chennai</i>	<i>Goa</i>	<i>Mumbai</i>	<i>Delhi</i>	<i>Bangalore</i>
Chennai	0	800	1280	2190	360
Goa	800	0	590	2080	570
Mumbai	1280	590	0	1540	1210
Delhi	2190	2080	1540	0	2434
Bangalore	360	570	1210	2434	0

An absolute lower bound for the TSP can be estimated as follows (Michalewicz and Fogel, 2004). For each row, add up the smallest two positive entries. In the above example, select 350 and 800 from row 1. The contribution of the two Chennai segments will be smallest when it lies between Goa and Bangalore in the tour. In this manner, pick the smallest two costs for each city, add them up and divide by two. For the above cost matrix, we get,

$$\begin{aligned}
 LB &= \frac{(360 + 800) + (570 + 590) + (590 + 1210) + (1540 + 2080) + (360 + 570)}{2} \\
 &= \frac{8670}{2} = 4335
 \end{aligned}$$

Observe that the above lower bound may not actually be feasible. This is because for each city (each row), we consider the nearest two neighbours as ones in the estimate. In the example map shown below in Figure 5.5, there are two long edges that must be part of any tour, but neither will figure in the lower bound estimate.

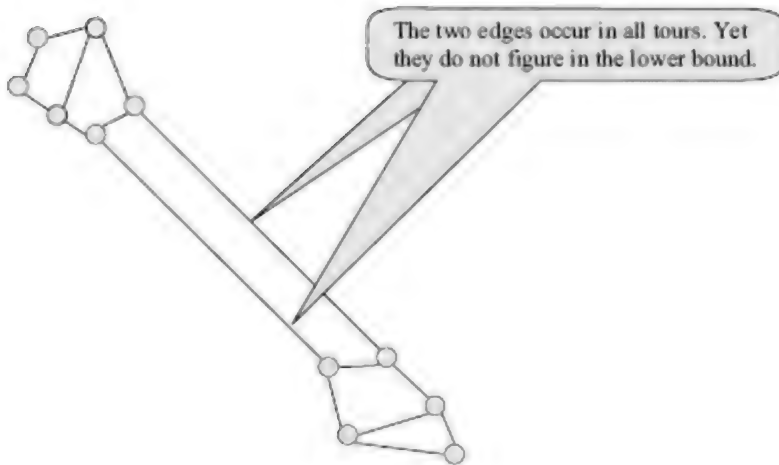


FIGURE 5.5 The lower bound estimated costs may not be feasible in practice.

Consider now the refinement search space. The root consists of a node representing all solutions. We can partition this set in two, one subset including a particular arc, and the other excluding it. These two sets can then be further refined recursively till each node describes one tour precisely. Part of the search tree is depicted below in Figure 5.6. It contains all the twelve distinct tours in the leaves. Some leaves that have not been fully refined contain more than one tour. A label like “CB” says the segment “CB” is included in the tour(s) in that node, while “¬CB” says that it is not present in the tour(s) in that node. In addition, the tours in a node must also be consistent with the labels at the ancestor nodes.

The next task is to estimate the cost of these partially refined solutions. Wherever an arc is known to be part of a tour, we add the known cost. Otherwise we choose the smallest possible segments. For example, the estimated cost of a tour including the *Chennai-Mumbai* (CM) segment is

$$LB = \frac{(360 + 1280) + (570 + 590) + (590 + 1280) + (1540 + 2080) + (360 + 570)}{2}$$

$$= \frac{9010}{2} = 4505$$

In the above case, the CM cost (1280) was included for both the Chennai and the Mumbai segments. The lower bound cost then went up to 4505. Suppose in addition to the above, we also want to include the *Mumbai Bangalore (MB)* segment. The cost of this segment, 1210, will have to be included as well.

$$LB = \frac{(360 + 1280) + (570 + 590) + (1210 + 1280) + (1540 + 2080) + (360 + 1210)}{2}$$

$$= \frac{10480}{2} = 5240$$

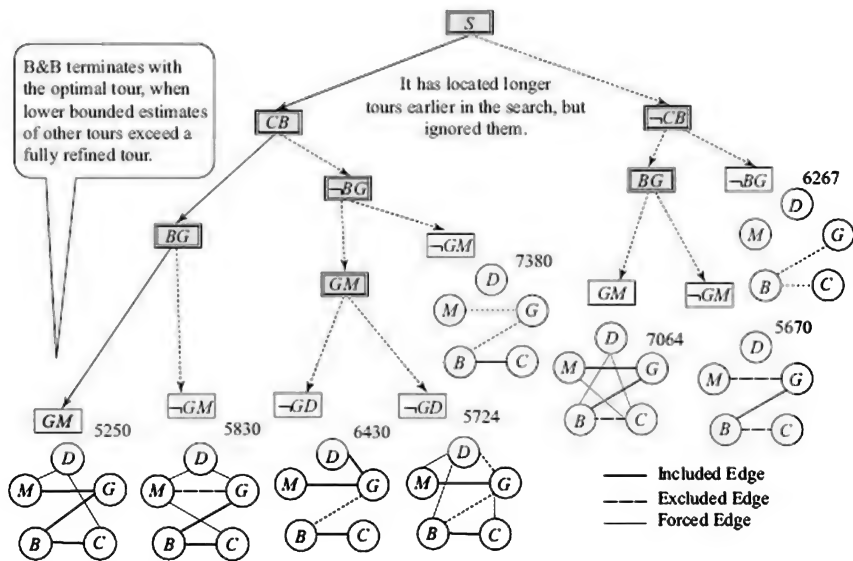


FIGURE 5.6 Branch and Bound on TSP in the refinement space.

While this does give us a tighter estimate, one can improve upon this with some reasoning. The fact that the CM segment and MB segments are included implies that the *Bangalore Chennai (BC)* segment with cost 360 should be excluded. Otherwise, we would have a C-M-B-C cycle, which is not allowed by the specification of TSP. The BC segment costing 360, contributes twice in the above estimate. One must replace the two occurrences with the next better costs, 800 and 570, as shown below.

$$\begin{aligned}
 LB &= \frac{(800 + 1280) + (570 + 590) + (1210 + 1280) + (1540 + 2080) + (570 + 1210)}{2} \\
 &= \frac{11130}{2} = 5565
 \end{aligned}$$

Both are incidentally to Goa, and therein lurks another case for reasoning, because each city can be connected only to two others. With further reasoning, we could get a still better estimate. This kind of trade off is not uncommon. Reasoning can prune the search space, but itself has an associated cost. In the above method for estimating costs for example, we may be counting more than two edges emanating out of a node. If we could avoid this, then our estimates would become more accurate.

As we refine the above partial solution, we get increasing and better estimates of cost. If at any point *all* the estimated costs were to become higher than a *known* cost of a complete solution in hand then we would stop refining the solution.

The *B&B* algorithm can thus be summarized as shown in Figure 5.7. The only condition is that the estimated cost must be a *lower bound* on the actual cost.

```

Generalized B&B Procedure()
1 Start with all possible solutions
2 repeat
3   Refine the least (estimated cost) solution further
4 until the cheapest solution s is fully refined
5 return s

```

FIGURE 5.7 The general *Branch & Bound* procedure.

Branch & Bound is a complete and admissible algorithm. That is, it will find a solution, if there exists one, and it is guaranteed to find an optimal solution. In terms of space and time complexity, however, it is not very good. The algorithm can be seen as a generalization of the Breadth First Search algorithm when each move has an associated cost. Like Breadth First Search, this algorithm too is uninformed and conservative, searching blindly without a sense of direction. Figure 5.8 shows an example map (to scale) where the *B&B* algorithm would spend a lot of time exploring nodes that are closer to the starting point before finding a path to the goal. Both the time and space complexity of *B&B* tends to be exponential.

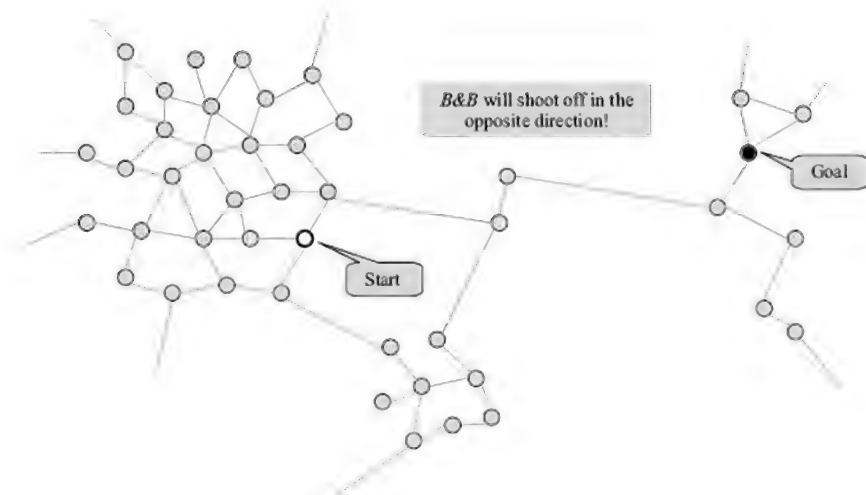


FIGURE 5.8 Branch & Bound has no sense of direction.

5.4 Dijkstra's Algorithm

Dijkstra's algorithm (DA) (Dijkstra, 1959; Cormen et al., 2001) is a well known shortest path algorithm on graphs. It solves a more general problem, known as the single source problem. The algorithm finds the shortest paths to all nodes in the graph from a given (source) node. In that sense, it is not concerned with reaching a specific goal node. The algorithm is briefly described in Figure 5.9. Observe, that one needs the complete graph for the algorithm, and it starts with colouring *all* the nodes white. A white node is one that is yet to be picked up by the algorithm. In every cycle, it picks one node, to which it has found the shortest path, and colours it black.

```

Dijkstra's Algorithm()
1 Colour all nodes white
2 cost(start) ← 0
3 parent(start) ← NIL
4 for all other nodes n
5   do cost(n) ← ∞
6 repeat
7   Select lowest cost white node n
8   Colour n black
9   for all white neighbours m of n
10    do if (cost(n) + k(n, m)) < cost(m)
11        then cost(m) ← cost(n) + k(n, m)
12            parent(m) ← n
13 until all nodes are coloured black
  
```

FIGURE 5.9 Dijkstra single source shortest path algorithm.

We illustrate the algorithm with our tiny search graph shown in Figure 5.10.

- Note that in the last iteration, a better path to G was found from E . The cost was updated from 11 to 10, and the parent pointer reassigned.
- The last node to be coloured is G , and the algorithm terminates. The shortest route to *any* node can be traced back.

5.5 Algorithm A*

The algorithm A^* , first described by Hart, Nilsson and Raphael, see (Hart et al., 1968; Nilsson, 1980) combines the best features of $B\&B$, Dijkstra's algorithm and *Best First Search* described earlier in Chapter 3.

Both $B\&B$ and Dijkstra's algorithm extend the least cost partial solution. While the latter is designed to solve a general problem, the former uses a similar blind approach, even though it has a specific goal to achieve. $B\&B$ generates a search tree that may have duplicate copies of the same nodes with different costs; while Dijkstra's algorithm searches over a given graph, keeping exactly one copy of each node and back pointers for the best routes. Neither has a sense of direction.

Best First Search does have a sense of direction. It uses a heuristic function to decide which of the candidate nodes is likely to be closest to the goal, and expands that. However, it does not keep track of the cost incurred to reach that node, as illustrated in Figure 5.11. *Best First Search* only looks ahead from the node n , seeking a quick path to the goal, while $B\&B$ only looks behind, keeping track of the best paths found so far. Algorithm A^* does both.

A^* uses an evaluation function $f(\text{node})$ to order its search.

$f(n)$ = Estimated cost of a path from Start to Goal *via* node n .

Let $f^*(n)$ be the (actual but unknown) cost of an optimal path $S \rightarrow n \rightarrow G$ as described above, of which $f(n)$ is an estimate. The evaluation function has two components as shown in Figure 5.12 below. One, backward looking, $g(n)$, inherited from $B\&B$, the *known* cost of the path found from S to n . The other, forward looking and goal seeking, $h(n)$, inherited from *Best First Search*, is the *estimated* cost from n to G .

$$\begin{aligned}f^*(n) &= g^*(n) + h^*(n) \\f(n) &= g(n) + h(n)\end{aligned}$$

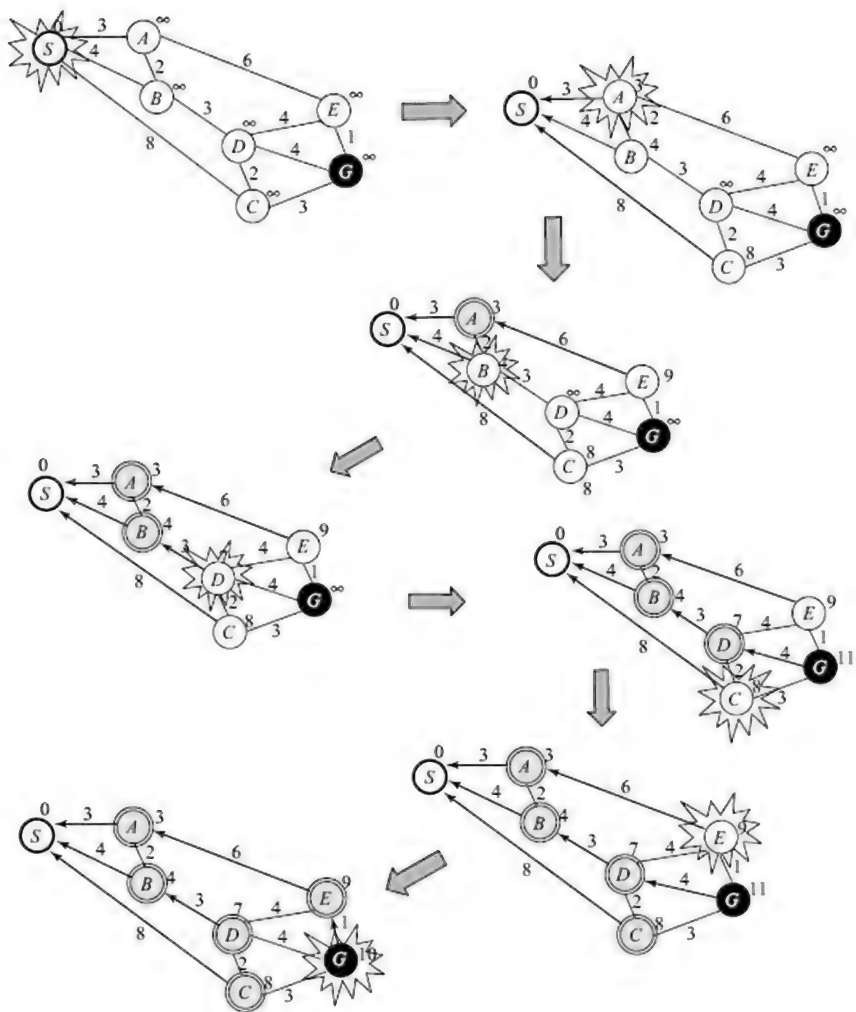


FIGURE 5.10 Dijkstra's algorithm on the tiny search graph.

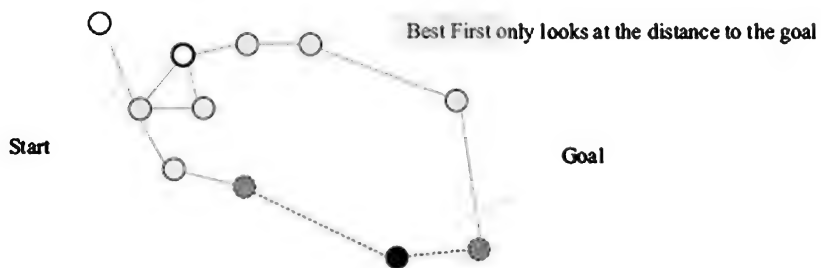


Figure 5.11 Best First chooses the node from *OPEN* closest to the goal. It may find a costlier path.

where $g^*(n)$ is the optimal cost from S to n , and $h^*(n)$ is the optimal cost from n to G . Note that $g^*(n)$ and $h^*(n)$ may not be known. What are known are $g(n)$ which can be thought of as an estimate of $g^*(n)$ that the algorithm maintains, and $h(n)$ the heuristic function that is an estimate of $h^*(n)$.

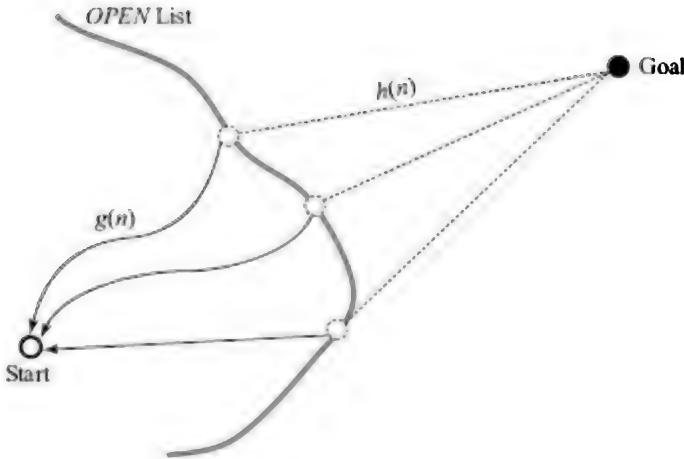


FIGURE 5.12 For all nodes, the function $f(n)$ is made up of two components, $g(n)$ and $h(n)$.

In general, $g^*(n)$ will be a lower than $g(n)$, because the algorithm may not have found the optimal path to n yet.

$$g(n) \geq g^*(n)$$

The heuristic value $h(n)$ is an estimate of the distance to the goal. In order for the algorithm to guarantee an optimal solution, it is necessary that the heuristic function *underestimate* the distance to the goal. That is,

$$h(n) \leq h^*(n)$$

We also say that $h(n)$ is a lower bound on $h^*(n)$. If the above condition is true then A^* is said to be *admissible*; that is, it is guaranteed to find the optimal path. We will look at a formal proof of the admissibility of algorithm A^* later in the chapter. Meanwhile, we illustrate with an example the intuition behind the condition that the heuristic function should underestimate the actual cost. Let an instance of A^* have two nodes, P and Q on the *OPEN* list, such that both are one move away from the goal. Let the cost of reaching both P and Q be the same, say 100. Let the actual cost of the move from P to G be 30, and let the cost of the move from Q to G be 40, as shown in Figure 5.13.

Let there be two versions of A^* , named A_1^* and A_2^* , employing two heuristic functions, $h_1(n)$ and $h_2(n)$. Let us assume that both have found the paths up to P and Q with cost 100. Let both heuristic functions erroneously evaluate Q to be nearer to the goal G than P is. But let A_1^*

overestimate the distance to G ; thus, in fact becoming inadmissible, while A_2^* underestimates the distance, as illustrated below.

$$h_1(P) = 50$$

$$h_1(Q) = 45$$

and

$$h_2(P) = 20$$

$$h_2(Q) = 15$$

Let us trace the progress of A_1^* first. Assuming that only P and Q are in the *OPEN* list, the f values are

$$f_1(P) = g_1(P) + h_1(P) = 100 + 50 = 150$$

$$f_1(Q) = g_1(Q) + h_1(Q) = 100 + 45 = 145$$

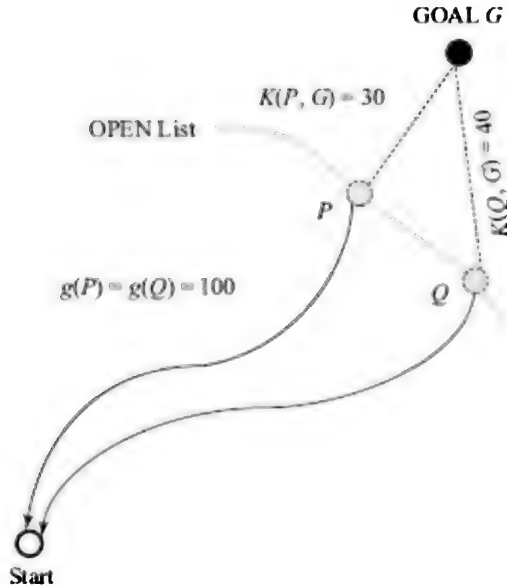


FIGURE 5.13 An instance of A^* nearing termination.

Since it has the smaller f -value, A_1^* picks Q and expands it generating the node G with $g_1(G) = 100 + 40 = 140$. It now has two nodes on *OPEN*, P and G with the values,

$$f_1(P) = g_1(P) + h_1(P) = 100 + 50 = 150$$

$$f_1(G) = g_1(G) + h_1(G) = 140 + 0 = 140$$

It now picks the goal G and terminates, finding the longer path with cost 140.

A_2^* too starts off by picking the node Q in the following position.

$$f_2(P) = g_2(P) + h_2(P) = 100 + 20 = 120$$

$$f_2(Q) = g_2(Q) + h_2(Q) = 100 + 15 = 115$$

It also finds a path to G with a cost of 140. For the next move, however, it picks P instead of G in the following position,

$$f_2(P) = g_2(P) + h_2(P) = 100 + 20 = 120$$

$$f_2(G) = g_2(G) + h_2(G) = 140 + 0 = 140$$

Thus, A_2^* picks P instead of G and finds the shorter path to G . This happened because it underestimated the cost of reaching the goal through P .

The algorithm A^* is described below. Like the Dijkstra's Algorithm, it uses a graph structure but one which it generates on a need basis during search. It is also called a *graph search algorithm*. It keeps track of the best route it has found so far to every node on the *OPEN* and *CLOSED*, via the *parent* link. Since it may find cheaper routes to nodes it has already expanded, a provision to pass on any improvements in cost to successors of nodes generated earlier, has to be made.

```

Procedure A*()
1  open ← List(start)
2  f(start) ← h(start)
3  parent(start) ← NIL
4  closed ← {}
5  while open is not EMPTY
6    do
7      Remove node n from open such that f(n) has the lowest value
8      Add n to closed
9      if GoalTest(n) = TRUE
10         then return ReconstructPath(n)
11     neighbours ← MoveGen(n)
12     for each m ∈ neighbours
13       do switch
14         case m ∉ open AND m ∉ closed :      /* new node */
15           Add m to open
16           parent(m) ← n
17           g(m) ← g(n) + k(n, m)
18           f(m) ← g(m) + h(m)
19
20         case m ∈ open :
21           if (g(n) + k(n, m)) < g(m)
22             then parent(m) ← n
23             g(m) ← g(n) + k(n, m)
24             f(m) ← g(m) + h(m)
25
26         case m ∈ closed :      /* like above case */
27           if (g(n) + k(n, m)) < g(m)
28             then parent(m) ← n
29             g(m) ← g(n) + k(n, m)
30             f(m) ← g(m) + h(m)
31             PropagateImprovement(m)
32 return FAILURE

PropagateImprovement(m)
1  neighbours ← MoveGen(m)
2  for each s ∈ neighbours
3    do newGvalue ← g(m) + k(m, s)
4    if newGvalue < g(s)
5      then parent(s) ← m
6      g(s) ← newGvalue
7      if s ∈ closed
8        then PropagateImprovement(s)

```

FIGURE 5.14 Algorithm A*.

The representation used here is different from the *nodePair* representation introduced in Chapter 2. Instead, an explicit *parent* pointer is maintained. This has been done because we want to keep only one copy of each node, and reassign parents when the need arises. Consequently, the definition of the *ReconstructPath* function will change. The revised definition is left as an exercise for the user.

In the following example, the node labelled *N* is about to be expanded. The values shown in the nodes are the *g* values. The double lined boxes are in the set *CLOSED* and the single lined ones in *OPEN*. Each node, except the start node, has a parent pointer. The dotted arcs

emanating from N show the successors of N , including a new node whose g value is yet to be computed.

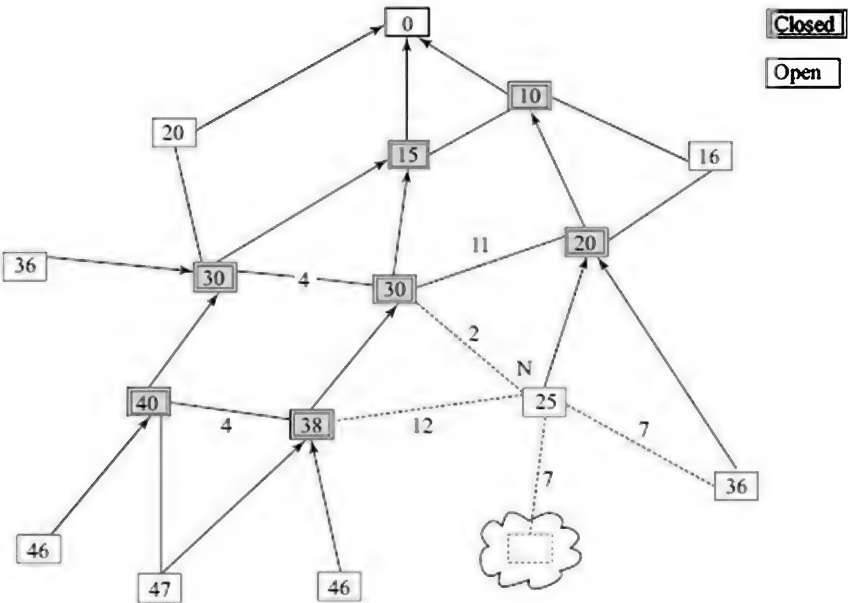


FIGURE 5.15 Node N is about to be expanded.

Figure 5.16 shows the changes that are made after the node N is expanded. Cheaper paths were found for some of the nodes on *OPEN*. Likewise, for some nodes on *CLOSED* too, and in their case the improved g -values had to be passed on to their descendents as well.

5.6 Admissibility of A*

The algorithm A^* will always find an optimal solution, provided the following assumptions (A1 – A3) are true.

- A1.** The branching factor is finite. That is, there are only a finite number of choices at every node in the search space.
- A2.** The cost of each move is greater than some arbitrarily small nonzero positive value ϵ . That is,

$$\text{for all } m, n: k(m, n) > \epsilon \tag{5.1}$$

- A3.** The heuristic function underestimates the cost to the goal node. That is

$$\text{for all } n: h(n) \leq h^*(n) \tag{5.2}$$

We prove the admissibility of algorithm A^* via a series of lemmas. We also prove that as the heuristic function becomes a better estimate of the optimal cost, the A^* search examines fewer nodes. The proofs are as given in (Nilsson, 1998). The proof is made up of a series of lemmas starting with L1 below.

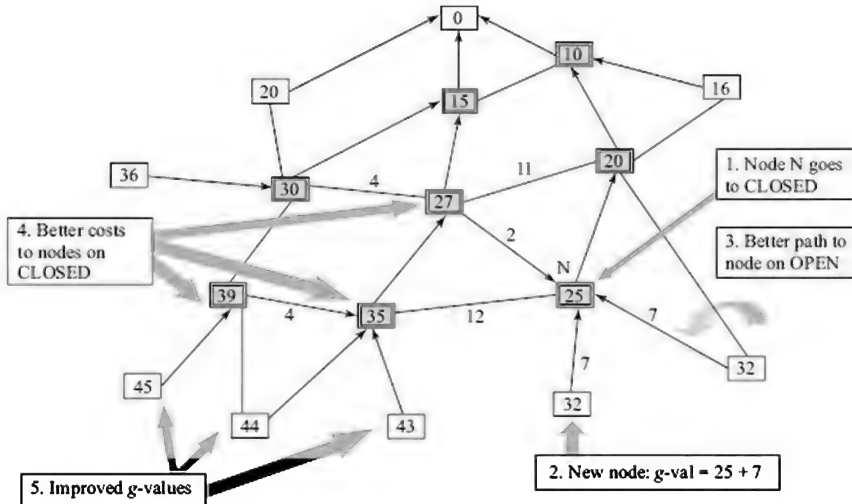


FIGURE 5.16 The graph after node N has been expanded.

L1: The algorithm always terminates for finite graphs.

Proof In every cycle of the main loop in A^* , the algorithm picks one node from *OPEN* and places it in *CLOSED*. Since there are only a finite number of nodes, the algorithm will terminate in a finite number of cycles, even if it never reaches the goal (that is, the goal is not reachable).

L2: If a path exists to the goal node then the OPEN list always contains a node n' from an optimal path. Moreover, the f-value of that node is not greater than the optimal cost.

Proof Let (S, n_1, n_2, \dots, G) be an optimal path as shown in Figure 5.17. To begin with, S is on *OPEN*. Node n_1 is a child of S . When S is removed from *OPEN*, n_1 is placed on *OPEN*. In this manner, whenever a node from the above path is removed from *OPEN*, the next node is placed on *OPEN*. And if G is removed from *OPEN* then A^* has terminated with the optimal path (S, n_1, n_2, \dots, G) to G .

Furthermore,

$$\begin{aligned}
 f(n') &= g(n') + h(n') \\
 &= g^*(n') + h(n') && \text{because } n' \text{ is on the optimal path } g(n') = g^*(n') \\
 &\leq g^*(n') + h^*(n') && \text{from (A3) } h(n') \leq h^*(n') \\
 &\leq f^*(n') \\
 &\leq f^*(S) && \text{because } n' \text{ is on the optimal path } f^*(n') = f^*(S) \\
 \therefore f(n') &\leq f^*(S)
 \end{aligned} \tag{5.3}$$

Note that $f^*(S)$ is the optimal cost path from S to G .

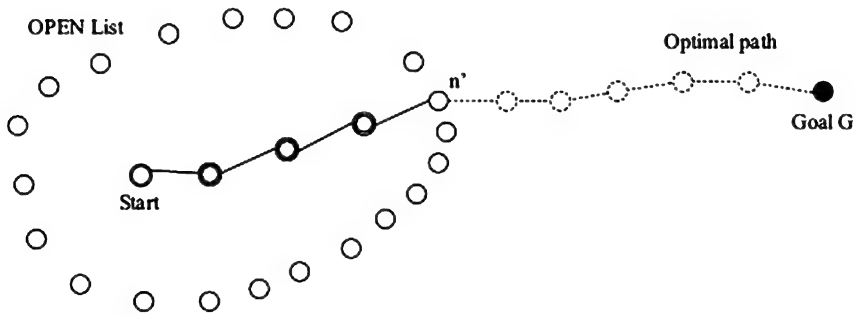


FIGURE 5.17 OPEN always contains a node from the optimal path.

L3: If there exists a path from the start node to the goal, A^* finds a path. This is true even if the graph is infinite.

Proof A^* always picks a node with the lowest f -value. Every time it extends a partial solution, the g -value of the partial solution increases by a finite value² greater than ϵ (A2). Also, since the branching factor is finite (A1), there are only a finite number of partial solutions cheaper than the cost of a path to the goal, that is $g^*(\text{Goal})$. After exploring all of them in a finite amount of time, eventually the path to the goal becomes cheapest, and is examined by A^* which terminates with a path to the goal.

L4: A^* finds the least cost path to the goal.

Proof (by contradiction)

Assumption A4: Let A^* terminate with node G' with cost $g(G') > f^*(S)$.

At the last step when A^* was about to expand G' , there must exist (L2) a node n' such that $f(n') \leq f^*(S)$. Therefore, $f(n') < f(G')$, and A^* would have picked n' instead of G' . Thus, assumption A4 is wrong, and A^* could not have terminated with any node with a suboptimal cost. Therefore, A^* terminates by finding the optimal cost path.

L5: For every node n expanded by A^* , $f(n) \leq f^*(S)$

Proof A^* picked node n in preference to node n' . Therefore,

$$f(n) \leq f(n') \leq f^*(S) \quad (5.4)$$

L6: A more informed heuristic leads to more focussed search.

Let A_1 and A_2 be two admissible versions of A^* using heuristic functions h_1 and h_2 respectively, and let $h_2(n) > h_1(n)$ for all n . We say h_2 is *more informed* than h_1 , because it is closer to the h^* value. Since both versions are admissible, both heuristic functions have $h^*(n)$ as the upper bound. Then any node expanded by A_2 is also expanded by A_1 . That is, the version with the more informed heuristic function is more focused on the goal, and will never generate more nodes than the less informed one.

Proof (by induction) We show that any node expanded by A_2 is also expanded by A_1 . The property P that we need to prove for all nodes n is,

$$\text{expands}(n, A_2) \Rightarrow \text{expands}(n, A_1)$$

which should be read as “if A_2 expands node n then A_1 expands node n ”.

Basis: $\text{expands}(S, A_2) \Rightarrow \text{expands}(S, A_1)$ since both start with S .

Hypothesis: Let P be true for all nodes up to depth k from S .

Proof step: (to show that property P is true for nodes at depth $k + 1$).

We do the proof step by contradiction.

Assumption: Let there exist a node L at depth $k + 1$ that is expanded by A_2 , and such that A_1 terminates without expanding node L .

Since A_2 has picked node L ,

$$f_2(L) \leq f^*(S) \quad \text{from (5.4)}$$

That is

$$g_2(L) + h_2(L) \leq f^*(S).$$

or

$$h_2(L) \leq f^*(S) - g_2(L) \quad (5.5)$$

Now, since A_1 terminates without picking node L ,

$$f^*(S) \leq f_1(L) \quad \text{because otherwise } A_1 \text{ would have picked } L$$

or

$$f^*(S) \leq g_1(L) + h_1(L)$$

or

$$f^*(S) \leq g_2(L) + h_1(L)$$

because $g_1(L) \leq g_2(L)$ since A_1 has seen all nodes up to depth k seen by A_2 , and would have found an equal or better cost path to L .

We can rewrite the last inequality as,

$$f^*(S) - g_2(L) \leq h_1(L) \quad (5.6)$$

Combining (5.5) and (5.6), we get,

$$h_2(L) \leq h_1(L)$$

which contradicts the given fact that $h_2(n) > h_1(n)$ for all nodes.

The assumption that A_2 terminates without expanding L is false, and therefore A_2 must expand L . Since L was an arbitrary node picked at depth $k + 1$, the property P is true for depth $k + 1$ as well. Thus, by induction for all nodes n , the property P is true. That is,

$$\text{For all nodes } n, \text{ expands}(n, A_2) \Rightarrow \text{expands}(n, A_1)$$

q.e.d.

The search space explored by the two functions is illustrated in Figure 5.18.

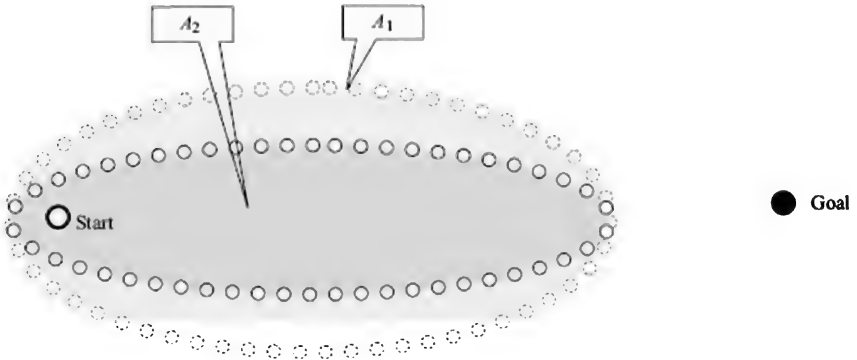


FIGURE 5.18 Search spaces for A_1 and A_2 when $h_2(n) > h_1(n)$.

5.6.1 The Monotone Property

The monotone property or the consistency property for a heuristic function says that for a node n that is a successor to a node m on a path

to the goal being constructed by the algorithm A^* using the heuristic function $h(x)$,

$$h(m) - h(n) \leq k(m, n) \quad (5.7)$$

That is, the heuristic function is such that it underestimates the cost of every segment individually along the way. Moving from node m to n , the reduction in heuristic value is less than the cost $k(m, n)$ incurred from m to n . Of course, this would be true, if node n were to be a bad successor of m , going for example in the opposite direction. The monotone property says that even if m and n were to be on an optimal path, this is true.

Given that we started from node S , we can rewrite the above inequality and add the term $g(m)$ to both sides to get,

$$h(m) + g(m) \leq k(m, n) + h(n) + g(m)$$

Since n is the successor of m , we have

$$g(m) + k(m, n) = g(n)$$

Therefore,

$$h(m) + g(m) \leq h(n) + g(n). \quad (5.8)$$

or

$$f(m) \leq f(n) \quad (5.9)$$

That is, on a path being constructed from S to G by A^* , the f -values increase as we move towards the goal G . This is true even for the optimal path. The closer you get to the goal, the better you estimate the actual cost, since the contribution of the heuristic component decreases. Remember that it is the heuristic function that underestimates the cost. Observe that the hill climbing algorithm using f -values would simply not work, because these values are increasing.

For A^* , the interesting consequence of searching with a heuristic function satisfying the monotone property is that every time it picks a node for expansion, it does so by finding an optimal path to that node. As a result, there is no necessity of improved cost propagation through nodes in *CLOSED* (lines 26–31 in Figure 5.14), because A^* would have already found the best path to them in the first place when it picked them from *OPEN* and put them in *CLOSED*.

L7: If the monotone condition holds for the heuristic function then at the time when A^* picks a node n for expansion $g(n) = g^*(n)$.

Figure 5.19 illustrates the situation. Let A^* be about to pick up node n with a value $g(n)$. Let there be a (known) optimal path from S to n via n_L and n_{L+1} .

Proof Let A^* expand node n with cost $g(n)$.

Let n_L be the last node on the optimal path from S to n that has been expanded. Let n_{L+1} be the successor of n_L that must be on *OPEN*. The following property holds,

$$h(n_L) + g(n_L) \leq h(n_{L+1}) + g(n_{L+1}) \quad \text{from (5.8)}$$

or

$$h(n_L) + g^*(n_L) \leq h(n_{L+1}) + g^*(n_{L+1})$$

because both are on the optimal path

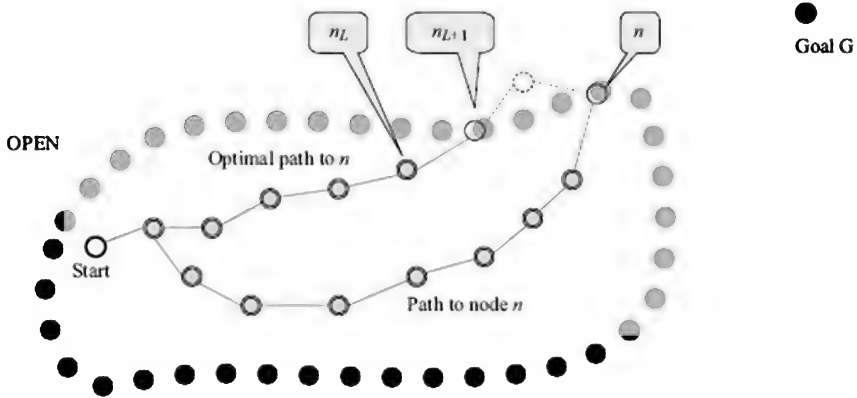


FIGURE 5.19 A^* is just about to pick node n with cost $g(n)$.

By transitivity of \leq , the above property holds true for any two nodes on the optimal path. In particular, it holds for n_{L+1} and node n . That is,

$$h(n_{L+1}) + g^*(n_{L+1}) \leq h(n) + g^*(n) \quad (5.10)$$

That is,

$$f(n_{L+1}) \leq h(n) + g^*(n) \quad \text{because } n_{L+1} \text{ is on the optimal path to } n$$

But since A^* is about to pick node n instead,

$$f(n) \leq f(n_{L+1})$$

That is,

$$h(n) + g(n) \leq f(n_{L-1})$$

or

$$h(n) + g(n) \leq h(n_{L+1}) + g^*(n_{L+1}) \quad (5.11)$$

Combining (5.10) and (5.11) we get,

$$h(n) + g(n) \leq h(n) + g^*(n)$$

$$\therefore g(n) \leq g^*(n)$$

$$\therefore g(n) = g^*(n) \text{ because } g(n) \text{ cannot be less than } g^*(n) \text{ the optimal cost.}$$

Therefore, whenever A^* expands any node, it does so after finding an optimal path to it.

q.e.d.

That means that nodes that are in *CLOSED* cannot have better paths to them, and consequently need not be updated.

5.6.2 Performance of Algorithm A^*

The algorithm A^* is complete and admissible. The results have been proven in the previous section. Both space complexity and time complexity of A^* are directly dependent on the heuristic function. With a perfect heuristic function, the algorithm will home in onto the goal in linear time with linear space requirement. In practice, heuristic functions are less than perfect, and the space and time requirements can be exponential, though with a lower branching factor than the actual one, as was discussed in the case of *Best First Search* earlier. Figure 5.18 above also illustrates the point that more accurate heuristic functions require lesser space, and, therefore, lesser time (since the size of *CLOSED* can be seen to be a measure of time). Incidentally, the figure can be visually misleading since we tend to see the search space as area inside the curve that grows only as quadratic. The reader must keep in mind that in many search spaces, the sizes of successive layers are multiples of preceding layers, leading to exponential growth. However, figures like Figure 5.18 above are useful for visualizing algorithms and we will continue to use them. We will also see problems where the growth in search space is quadratic, which are better illustrated by these figures, and in which there is a combinatorial growth of choices.

Time complexity by itself can only improve with a better heuristic function, or at the expense of admissibility. For example, many researchers experiment with a variation known as *Weighted A^** which uses the following function to order the search nodes,

$$f(n) = g(n) + k * h(n)$$

The factor k is used to control the pull of the heuristic function. Observe that as k tends to zero, the algorithm is controlled by $g(n)$ and it tends to behave like *Branch & Bound*. On the other hand, as we choose larger and larger values of k , the influence of $h(n)$ on the search increases more and more, and the algorithm tends to behave more like *Best First Search*. With values of k greater than one, the guarantee of finding the optimal solution goes, but the algorithm explores a smaller portion of the search space.

The issue of space complexity can be addressed, though at the cost of additional time. We look at some algorithms that require much lower space in the following sections.

5.7 Iterative Deepening A* (IDA*)

Algorithm *IDA** (Korf, 1985a) is basically an extension of *DFID* algorithm seen earlier in Chapter 2. *IDA** is to *A** what *DFID* was to *DFS*. It converts the algorithm to a linear space algorithm, though at the expense of an increased time complexity. It capitalizes on the fact that the space requirements of *Depth First Search* are linear. Further, it is amenable to parallel implementations, which would reduce execution time further. A simple way to do that would be to assign the different successors to different machines, each extending different partial solutions. The *IDA** algorithm is described below in Figure 5.20. The algorithm uses a search bound captured in a variable named *cutoff*. The initial value of *cutoff* is set to the lower bound cost, as seen from the start node S . Since this is a lower bound, any solution found within *cutoff* cost must be optimal. The observant reader would have noticed that it is quite unlikely that the solution would be found in the first iteration when $cutoff = f(S)$. This is because the heuristic function is designed to underestimate the optimal cost. However, if the *DFS* search fails, in the next iteration the cutoff value is incremented to the next lowest f -value from the list *OPEN*. In this way, the value of *cutoff* is increased incrementally to ensure that in any iteration, only an optimal cost solution can be found.

```

IDA* ()
1  cutoff ← f(S) = h(S)
2  while goal node is not found or no new nodes exist
3    do use DFS search to explore nodes with f-values within cutoff
4    if goal not found
5      then extend cutoff to next unexpanded value if there exists one

```

FIGURE 5.20 Algorithm Iterative Deepening A*.

While the algorithm *IDA** essentially does *Depth First Search* in each iteration, the space that it explores is biased towards the goal. This is because the f -value used for each node is the sum of the g -value and the h -value. As for paths that are leading away from the goal, the h -values

will increase and such paths would be cut off early. Thus, while DFS itself is without a sense of direction, the fact that f -values are used to prune the search pulls the overall envelope that it searches within towards the goal node as depicted in Figure 5.21.

Like DFS, the space required of IDA^* grows linearly with depth. We do not need to maintain a *CLOSED* list if we keep track of the solution path explicitly. There is, however, a drawback that in problems like city map route, finding the algorithm may expand internal nodes many times. This happens because there are many routes to a node, and each time the node is expanded all over again. For large problem sizes, this can become a problem. Figure 5.21 below illustrates the search space explored by IDA^* with some value of *cutoff*. One can see that there are combinatorially many paths to any node within the cutoff range. In the absence of a *CLOSED* list, IDA^* will visit all nodes through all possible paths. Nevertheless, for many problems, IDA^* can be a good option, specially if the number of combinations are small.

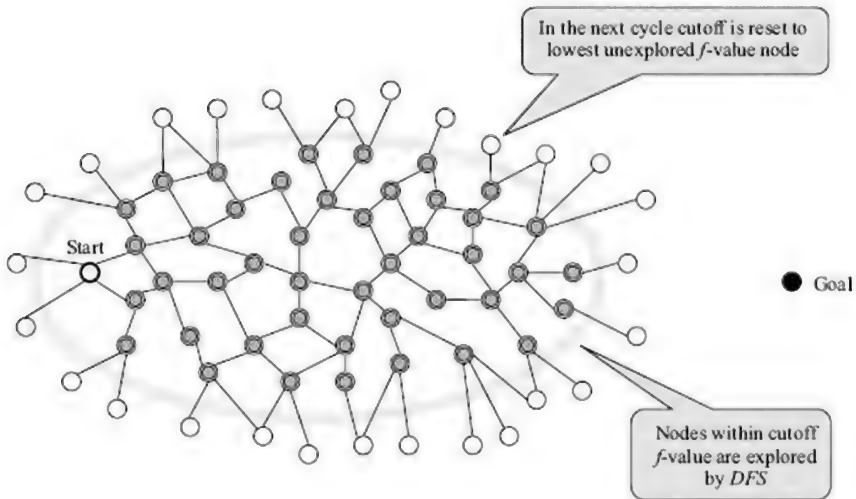


FIGURE 5.21 IDA^* iteratively extends search frontier for Depth First Search.

A factor that may adversely affect running time is when the increment in the cutoff value is such that only a few more nodes are included in each cycle. In the worst case, only one node may be added in each iteration. While this is necessary to guarantee admissibility, one could trade off execution time with some controlled loss in the solution cost. For example, one could decide in advance that the cutoff bounds will be increased by a value δ that is predetermined. The loss then will be bounded by δ , and an appropriate choice may be made for a given application. Figure 5.22 illustrates this situation. In the illustration, two goal nodes come within the ambit of *cutoff* when it is increased by δ . However, since the underlying search is DFS, it may terminate with the

more expensive solution.

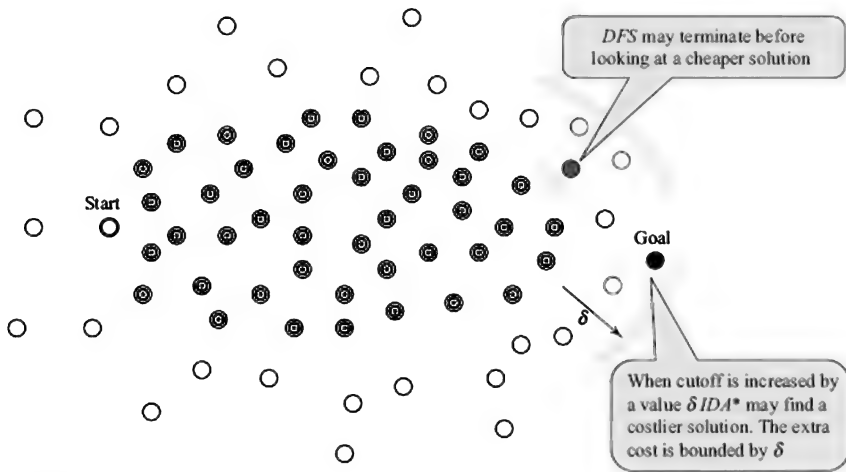


FIGURE 5.22 Loss in optimality is bounded by δ .

5.8 Recursive Best First Search (RBFS)

One thing that *IDA** suffers from is a lack of sense of direction, since the algorithm searches in a depth-first manner. Another algorithm developed by Richard Korf called *Recursive Best First Search (RBFS)* also requires linear space, but uses backtracking (Korf, 1993). One can think of the algorithm as heuristic depth-first search in which backtracking occurs when a given node does not have the best *OPEN* node amongst its successors. The interesting feature is that having explored a path once, if it backtracks from a node, it remembers the *f*-values it has found. It uses backed up *f*-values to update the values for nodes it has explored and backtracked on. The backed up value of a node is given by,

$$f_{\text{backed-up}}(\text{node}) = f(\text{node}) \quad \text{if node is a leaf node} \\ = \min \{f_{\text{backed-up}}(\text{child}) \mid \text{child is a successor of node}\} \quad \text{if node is not a leaf node}$$

Figure 5.23 depicts the behaviour of *RBFS*. As shown in the figure on the left, it pursues the middle path from the root node with heuristic value 55, till it reaches a point when all successors are worse than 59, its left sibling. It now rolls back the partial path, and reverts to the left sibling with value 59. It also revises the estimate of the middle node from 55 to 60, the best backed-up values as shown by the upward arrows.

From the point of implementation, *RBFS* keeps the next best value (in Figure 5.23, this is 59) as an upper bound for search to progress further. Backtracking is initiated when all the children of the current node become costlier than the upper bound. Observe that like *DFS*, it maintains only one path in its memory, thus requiring linear space. Its time complexity is

however difficult to characterize. One can imagine that in a large search space, it will often switch attention, sometimes even between the same two paths. This has been corroborated by experimental results, where it was found to take considerably longer solving problems, specially those where even though the problem only grows polynomially, the number of different paths grow combinatorially. Like *IDA**, *RBFS* ends up repeatedly visiting the same nodes again and again, leading to an increase in the computation time. This is the price both the algorithms have to pay for saving on space.

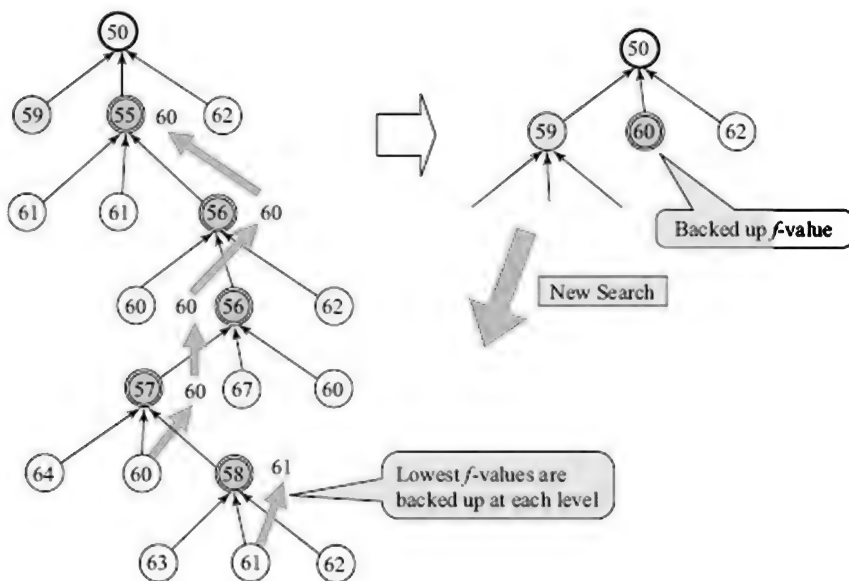


FIGURE 5.23 Recursive Best First Search rolls back a path when it is not looking the best.

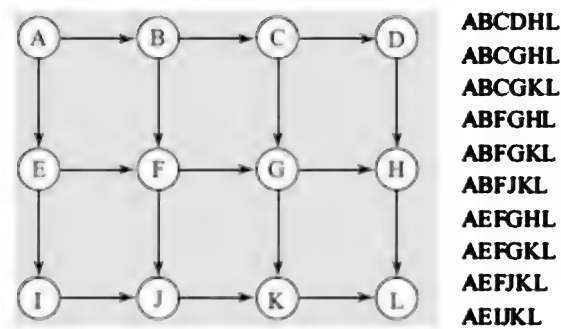
We now explore some other approaches to reducing memory, starting with the list *CLOSED*.

Maintaining the *CLOSED* list has two benefits. One, that it keeps a check on the nodes already visited, and prevents the search from expanding them again and again. And two, it is the means for reconstructing the path after the solution is found. In the next section, we look at an algorithm that prunes the *CLOSED* list, and still does both but, again, at the expense of more time complexity.

Box 5.1: When is *CLOSED* a problem?

One would expect that the size of the *CLOSED* list should be less of a concern than the size of *OPEN*, given the fact that in an exponentially growing search space, the *OPEN* is likely to be much bigger than *CLOSED*. There are, however, problems where the list *CLOSED* could be the main memory bottleneck. These are

problems that are combinatorial in nature. This happens when the underlying search space is a graph, for example in the route finding problem illustrated here.



One can see that there are many paths from node *A* to node *L*. The number of paths in an $n + 1$ by $m + 1$ rectangular grid is $(n + m)! / (n!m!)$. While the problem size increases only as a quadratic, algorithms like *DFS* or *IDA** will have to explore combinatorially many paths. A 25×25 grid would have 10^{14} paths! Moreover, the size of the *CLOSED* grows as a quadratic, while the size of *OPEN* will only increase linearly. If one were to tackle large problems, then it is *CLOSED* that will require more memory. Thus, in algorithms that will prune, *CLOSED* will be valuable.

Recently, problems thrown up by the field of computational biology have precisely this kind of domain (Korf et al., 2005; Korf and Zhang, 2000). For example, the alignment of two DNA sequences can be posed as a path finding in a grid like the one above, with additional diagonal arcs. In sequence alignment problems, one is allowed to insert gaps in the sequences. For example, the sequences *ACGTACGTACGT* and *ATGTCGTCACGT* can be best aligned with gaps as

ACGTACGT — *ACGT*
ATGT— *CGTCACGT*

(The worst is *ACGTACGTACGT* -----
 --- - - - - - *ATGTCGTCACGT*)

Notice that the second characters do not match. If the cost of each mismatch is 1, and the cost of a gap insertion is 2 then the cost of the best solution (alignment) is 5. The problem can be mapped to the problem of traversing a two dimensional grid with diagonal paths included (Needleman and Wunsch, 1970). The two given sequences form the two axes of the grid. Horizontal and vertical moves correspond to gap insertions. Diagonal moves correspond to character alignment and may have a cost of zero or

one, depending upon whether the two characters are the same or different.

Further, the sequence alignment problem can easily be extended to multiple sequences. An n -sequence problem will form an n -dimensional grid.

5.9 Pruning the *CLOSED* List

One of the functions of the *CLOSED* list is to prevent nodes from being expanded again and again. However, for a node on *CLOSED* to be expanded again, it will have to be generated as a child of some node on *OPEN* that is being expanded. In the *DFS* search algorithms discussed earlier (Figure 2.18), a function *removeSeen* prevented old nodes from being added to *OPEN* again. However, one can also prevent the search from regenerating the *CLOSED* nodes again by observing that for search to “leak back” into *CLOSED* it will have to go through nodes that are children of nodes on *OPEN*. For any node that is being expanded, it suffices to exclude successors that are “behind” the node. An important condition that must be satisfied is the consistency condition. As shown in Section 5.6.1, the consistency condition implies that all nodes in *CLOSED* have the best path to them discovered already. If that is the case then in the *A** algorithm there is no need to generate nodes already in the *CLOSED* again.

5.9.1 Divide-and-Conquer Frontier Search

One way of doing this is to modify the moves that can be made from the nodes on *OPEN* to keep a “tabu” list of disallowed successors for each node that is added to *OPEN*. The move generator is modified such that every time a node X (on *OPEN*) is generated as a successor of some node Y , Y is excluded from becoming a successor of X . If the node X is generated later as a successor of some other node Z then Z is also excluded from its list of successors of X . That is, every time a node is expanded, it is put on a tabu³ list of all *its* successors. And when a node is expanded, only the non-tabu successors are generated. As a consequence, every arc in the search graph is traversed only once, and only in one direction. In this way, search can be constrained to only move “forward”. An algorithm that uses this approach is called *Divide-and-Conquer Frontier Search (DCFS)* (Korf and Zhang, 2000). Along with every node on *OPEN*, the algorithm *DCFS* keeps a list of disallowed moves. The list *CLOSED*, therefore, is no longer needed to prevent the search from leaking back.

The second task of reconstructing the path when the goal is found still remains. *DCFS* addresses this problem by storing a relay node around the halfway mark in the search space for every node on *OPEN*. The

halfway point could be approximately determined when the *g-value* is close to the *h-value* for a node. Every node on *OPEN* that is beyond the halfway mark keeps a pointer to *its* relay node *Relay*, as shown in the Figure 5.24. Note that different nodes on *OPEN* may have different relay nodes.

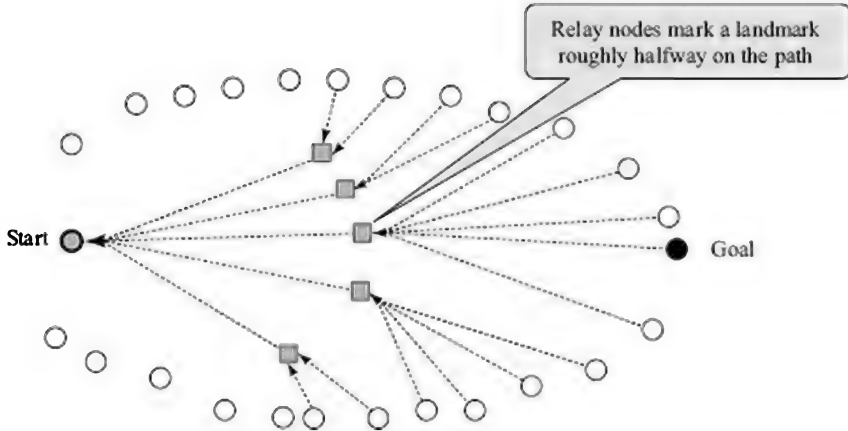


FIGURE 5.24 Divide-and-Conquer uses relay nodes to remember landmarks.

When the *DCFS* algorithm picks the goal node, it knows the cost of the solution, but not the path. It has only a pointer to the relay node (*Relay*) on its path, roughly on the halfway mark. To determine the path, *DCFS* is recursively called twice; once to find the path from *Start* to *Relay*, and next to find the path from *Relay* to *Goal*. It has divided the search problem into two parts. This process of recursive calls continues, till the entire path is reconstructed. The algorithm thus saves on space at the expense of running time. If $T(d)$ is the time required to search a path of length d then *DCFS* has a time complexity given by,

$$T(d) + 2 * T(d/2) + 4 * T(d/4) + \dots + k * T(d/k), \text{ where in the last term, } d/k = 1.$$

The reader can verify that the above sums up to $T(d) * \lg(T(d))$. In the specific case where $T(d)$ is b^d with branching factor b , this becomes $O(b^d * d)$. That is, if one were to do divide-and-conquer reconstruction when search is exponential, one has to do an equivalent of d searches instead of one.

The basic idea of divide and conquer solution reconstruction was adopted from similar techniques used in dynamic programming methods developed earlier for sequence comparison (Hirschberg, 1975; Myers and Miller, 1988) before memory constraints led researchers to look at *A** and its variants.

5.9.2 Sparse-Memory Graph Search

A variation for pruning of the *CLOSED* list takes a different approach. The *Sparse-Memory Graph Search (SMGS)* identifies the *boundary* of the *CLOSED* list, as shown in Figure 5.25 (Zhou and Hansen, 2003). The boundary can be defined as those nodes on *CLOSED* that have at least one neighbour (successor) still on *OPEN*. This can be done by keeping a counter with every node when it is expanded to keep track of the number of children it has on *OPEN*. The counter is decremented each time its children are expanded (the node will appear as a child). As long as the counter is greater than zero, the node is on the boundary. When it becomes zero, it goes into the *kernel*. The nodes of *CLOSED* that are not on the boundary are in the kernel. One can observe that the nodes in the kernel can only be reached via the nodes on the boundary. It would thus be enough to check for new successors to be on the boundary, to prevent the search from leaking back. The nodes in the kernel can be pruned away.

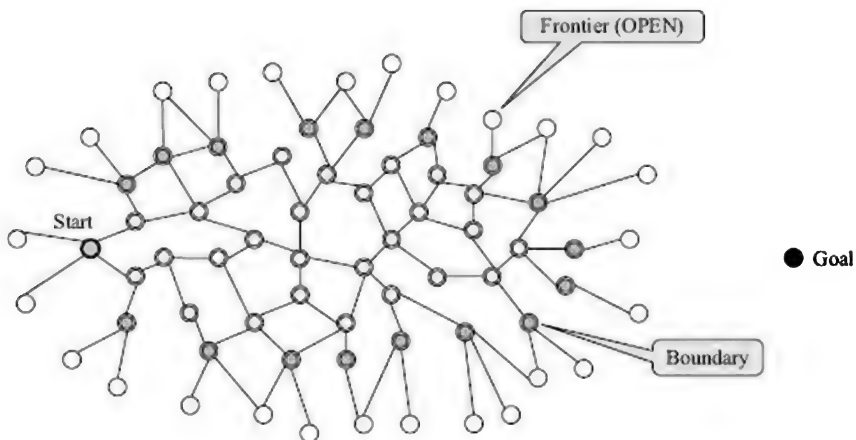


FIGURE 5.25 Boundary nodes in the *CLOSED* list are enough to prevent search from “leaking” back.

The *SMGS* also keeps relay nodes for reconstruction of the path like the *DCFS*. The difference is that it calls a module *PruneClosed* to prune the *CLOSED* list only when it senses that it is running out of memory.

The algorithm identifies three kinds of nodes. One, the *kernel* nodes that have been inspected and all their successors have been inspected. And two, the *boundary* nodes that have been inspected but have some successors on *OPEN*. Finally, the nodes in *OPEN* are the ones that have been generated but have not been inspected. Together, the *kernel* and the *boundary* would form the *CLOSED* set. Initially, the *Start* node is marked as a *relay* node.

The algorithm begins by keeping all three kinds of nodes, and

proceeds to pick nodes from *OPEN* and inspect them. Then at any time if it senses that it is running out of memory, it does the following, by calling a *PruneClosed* function. First, for every node on *OPEN*, it marks the corresponding nodes on the *boundary* as *relay* nodes. Then for each node on the *boundary*, it traces the back pointers to the latest *relay* node, and sets an ancestor pointer to that *relay* node. Then it deletes all *kernel* nodes that are not *relay* nodes. Having finished the pruning, it continues to pick nodes from *OPEN* and inspect them. The first time this is done, the ancestor pointer points to Start. The process is illustrated in Figures 5.26 and 5.27.

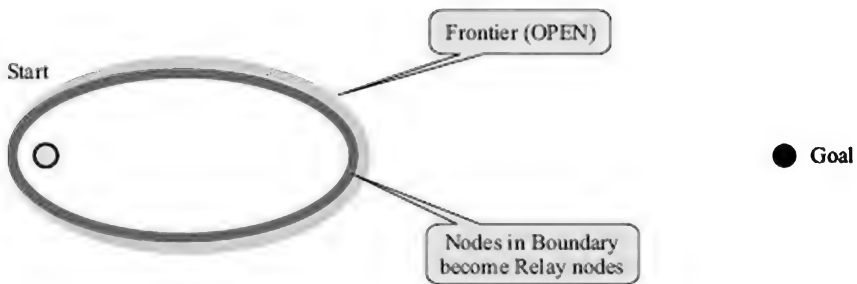


FIGURE 5.26 SMGS converts boundary nodes into Relay when a prune module is called.

The SMGS algorithm may create more than one relay layer if it is solving a large problem. It would have to recursively solve each segment between two consecutive relay nodes; otherwise it continues like the *A** search algorithm. Therefore, while on the one hand it does pruning only when necessary, thus saving on reconstruction cost; on the other hand, it can create many relay layers, thus being able to tackle larger problems.

When the search terminates, there may be several relay layers in the memory. In Figure 5.27 below, we illustrate this with two relay layers. Thus, at the point when search picks the goal node, it also has a Sparse Solution Path to the Start node via the relay nodes. Like the *DCFS* algorithm, *SMGS* recursively calls itself with each segment in the Sparse Solution Path to find the Dense Solution Path, the solution required. While *DCFS* divides the problem into two parts, *SMGS* may divide it in many parts, depending upon how many times the module *PruneClosed* is called. In Figure 5.27, the problem has been divided into three segments, one of which has not yet been pruned.

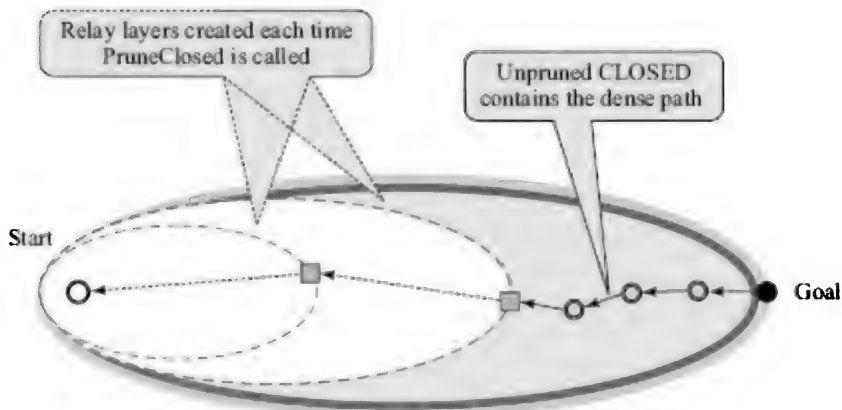


FIGURE 5.27 On termination, *SMGS* has a Sparse Solution Path to the goal.

Of the two search algorithms seen above, *DCFS* prunes *CLOSED* as it goes along. The only nodes it keeps are one relay node for each node on *OPEN*. It does so by modifying the nodes on *OPEN* to keep only forward-going successors. *SMGS* does not tamper with *OPEN*. Instead, it keeps a boundary layer amongst the nodes in *CLOSED* to prevent search from turning backwards. When a call to *PruneClosed* is made, this boundary becomes a relay layer. It also introduces the tactic of pruning only when memory is sensed to be running out, and keeps the option of pruning more than once and creating many relay nodes.

5.10 Pruning the *OPEN* List

In the previous section, we looked at a method to prune the *CLOSED* list. We also observed that it was useful for problems where the problem space only grew polynomially with depth, in which case the list *OPEN* is generally much smaller, often only growing linearly. However, in general, when problem space grows exponentially, it is the *OPEN* list that accounts for most of the memory. In this section, we look at ways to prune the *OPEN* list.

5.10.1 Breadth First Heuristic Search

In the chapter on State Space Search (Chapter 2), we had observed that *Breadth First Search* suffered from an exponentially growing memory requirement. The main reason for that was that the search was uninformed. If we can somehow generate an upper bound U on the solution cost then we could prune away nodes whose f -values are higher than U . This is because f -values are known to be lower bound estimates of solutions containing that node. The upper bound estimate could itself be obtained by using an inexpensive method like *Beam Search*, using

only the heuristic function $h(n)$. The resulting search algorithm called *Breadth First Heuristic Search (BFHS)* (Zhou and Hansen, 2004) has been shown to use less memory than A^* search. It explores nodes in a breadth first manner, but prunes nodes that have the estimated cost $f(n)$ larger than the upper bound U . The following figure suggests why the algorithm keeps a smaller *OPEN* list, and it can be seen that the better the heuristic function, the tighter will be the upper bound.

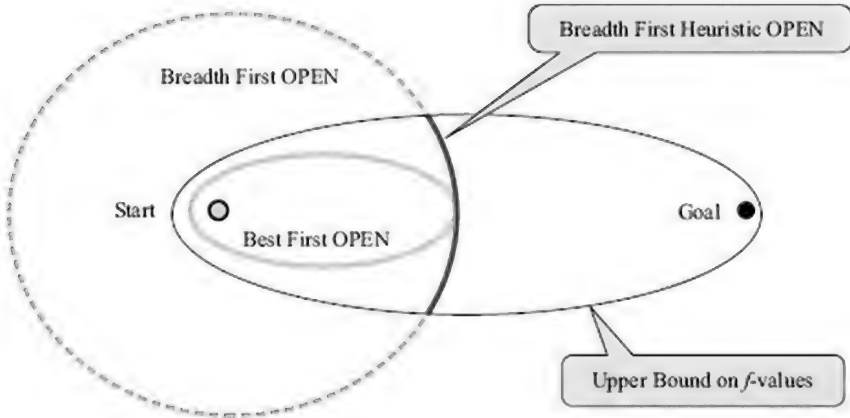


FIGURE 5.28 Breadth First Heuristic Search prunes the *OPEN* using f -values.

Observe that the pruning of nodes from the *Breadth First* frontier is admissible because the pruned nodes cannot be part of the optimal solution. Thus, *BFHS* is a variation that is complete and admissible. It requires lower memory for *OPEN* than A^* , but may expand more nodes than A^* does. The memory required by the *OPEN* of *BFHS* peaks somewhere around the halfway mark. After that point, the number of nodes that have f -values within the upper bound starts decreasing. This happens because as one comes closer to the goal, the contribution of the heuristic function $h(n)$ to the f -values becomes smaller, and the f -values become more and more accurate.

5.10.2 Divide and Conquer Beam Search

Beam Search (using f -values) can be seen as further pruning the *OPEN* list as shown in Figure 5.29 below. Given that the consistency condition entails that f -values increase when one proceeds towards the goal, one has to relax the criterion of moving to only better nodes, and instead move to the best w nodes at each level. Since it uses a constant beam width w , the memory requirements of *Beam Search* will grow only linearly with depth.

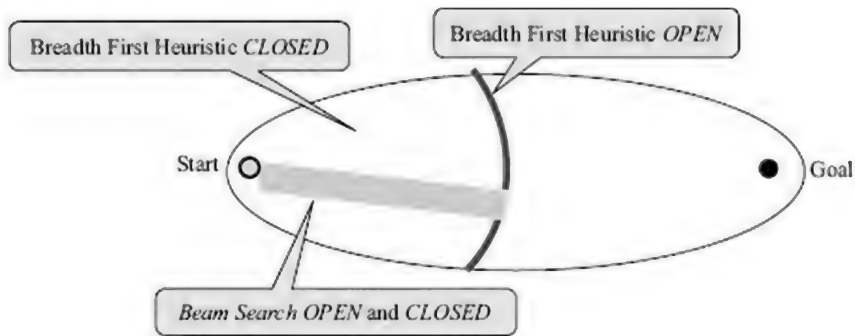


FIGURE 5.29 *Beam Search* further prunes the *OPEN* to a constant width w .

The memory required by *Beam Search* is proportional to dw , where d is the depth and w the beam width. However, *Beam Search* is inadmissible, not guaranteeing an optimal solution. In fact, it is not even guaranteed to find any solution. Later, we will address this issue and look at a way to make it complete.

Meanwhile, both *BFHS* and *Beam Search* keep the full *CLOSED* list. If we apply the techniques of pruning the *CLOSED* list from the last section to these two search methods, we get *Divide-and-Conquer BFHS* (*DCBFHS*) and *Divide-and-Conquer Beam Search* (*DCBS*). *Beam search* expands nodes in a manner similar to breadth first search, except for sorting and pruning each layer to a fixed number of nodes before proceeding to the next. Both may have a partially expanded layer containing both nodes on *OPEN* and *CLOSED*. The next layer will contain a partially formed *OPEN*, while the preceding layer will have only *CLOSED* nodes. The current and the preceding layer will contain the boundary of the search, and are enough to prevent the search from leaking back. In addition, the divide and conquer strategy requires a *RELAY* layer around the halfway point, to which nodes on *OPEN* will hold a pointer to for path reconstruction. Before the search has crossed the halfway mark, the pointer will point to the *Start* node. These four layers are enough for search to progress without regenerating nodes in *CLOSED*. Figure 5.30 illustrates the four layers.

Observe that *Divide-and-Conquer Beam Search* keeps a maximum of w elements in each of the four layers. Thus, its memory requirement is $4w$, which is a constant amount! That means that using *DCBS*, one can search up to any depth using a constant amount of memory. As a corollary, it allows us to fix the beam width w as high as resources will allow. The only problem is that it is incomplete.

Next, we look at an approach that allows *Beam Search* to backtrack and try other paths systematically, giving us a complete and admissible search algorithm.

5.10.3 Beam Stack Search

Beam Stack Search (BSS) is essentially beam search with backtracking. One of the reasons for the poor time performance of *IDA** and *RBFS* is that they pursued only one path at a time. In some sense, they underutilised the space available to the algorithm. *Beam Search* allows one to pursue multiple candidates simultaneously, but like *Hill Climbing*, cannot switch paths midcourse, and is therefore incomplete.

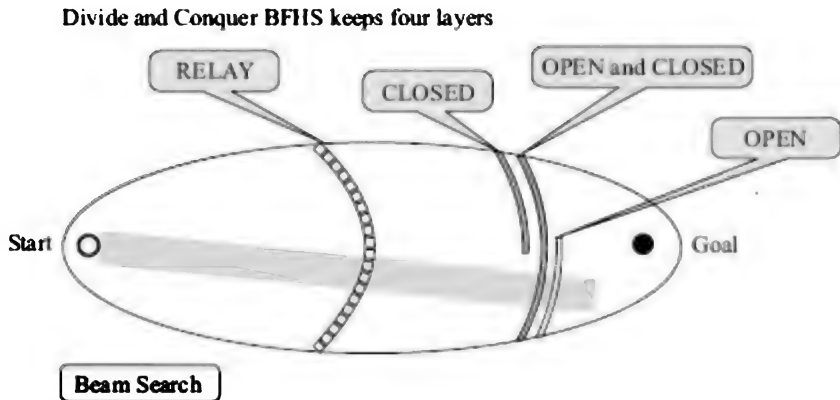


FIGURE 5.30 Divide and Conquer *Beam Search* keeps four layers of constant width.

Beam Stack Search (Zhou and Hansen, 2005) explores the search space *systematically* with a beam of width w . Like the *Beam Search*, it too may prune nodes inadmissibly, but it retains the option to backtrack to explore those nodes later. It does this by sorting the nodes at each level on their f -values, and keeping track of the minimum and maximum f -values of nodes admitted in the beam, at each stage of the search. It does this by keeping a separate stack, called the *Beam Stack*, in which it stores the f_{\min} and f_{\max} values at each level as a pair $[f_{\min}, f_{\max})$ as shown in Figure 5.31 below. Since the algorithm involves sorting of nodes at each level, it is easier to visualise with the search tree it generates.

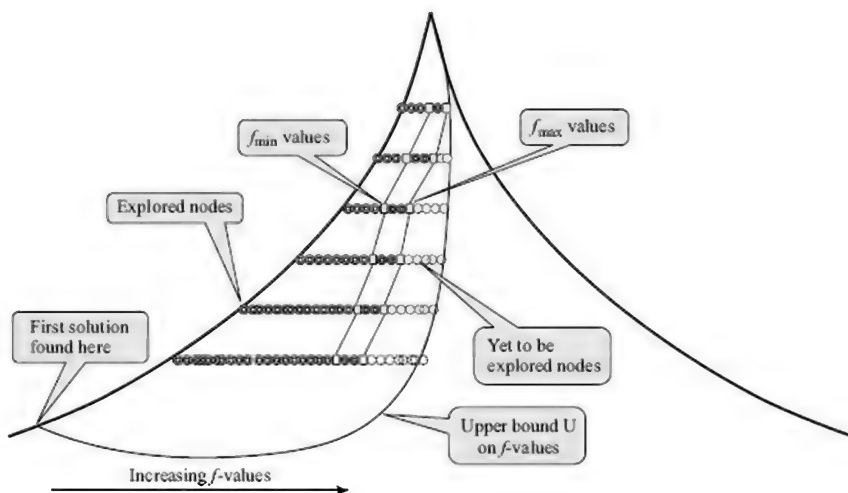


FIGURE 5.31 Beam Stack Search orders nodes on f -values. It slides the f_{\min} , f_{\max} window on backtracking, but does not go beyond the upper bound U .

The beam stack maintains the f_{\min} and f_{\max} values at each level. This corresponds to the window in that layer that is exposed to the beam search. The f_{\max} value corresponds to the lowest f -value node that was pushed out of the beam. When the algorithm backtracks, it slides the corresponding window by setting the new f_{\min} to the old f_{\max} . It sets the new f_{\max} value to the upper bound U , updating it to a lower value, only if nodes are generated that cannot be accommodated in the beam width. In this way, it can systematically explore the entire space that A^* would have explored.

Initially, it behaves like *Beam Search*. When it finds a solution, it sets the upper bound value, if a cheaper solution is found. This updating happens every time *BSS* finds a better solution. Once it has found a solution, the solution can be called for at any time. Thus, *BSS* becomes an *anytime algorithm*, quickly finding a good solution which can be returned on demand, but continuing to explore the rest of the space looking for better solutions. The best solution so far can be returned on demand. Moreover, since it uses a heuristic function and explores the lowest f -value nodes first, it is quite likely to find the best solution early.

Beam Stack Search is a complete and admissible version of *Beam Search*. Like *Beam Search* (with the *CLOSED* list), the space required by the algorithm is linear with depth.

Finally, the divide and conquer strategy can be applied to *Beam Stack Search* as well, to give a complete and admissible algorithm that requires almost constant space.

5.11 Divide and Conquer Beam Stack Search

Like *DCBS*, the *Divide and Conquer Beam Stack Search (DCBSS)* (Zhou and Hansen, 2005) also keeps four layers of width w in memory, as shown in Figure 5.32. Like the *BSS*, it maintains a beam stack that marks the f_{\min} and f_{\max} values for each layer. Strictly speaking, the space requirements of *DCBSS* are not constant because of the beam stack, which grows linearly with depth. But since only two values are stored per layer, the space requirements are presumably much smaller than those required to encode the state for each node. The number of *nodes* that *DCBSS* keeps are however constant, being $4w$, and thus the algorithm can permit a suitably large beam width w .

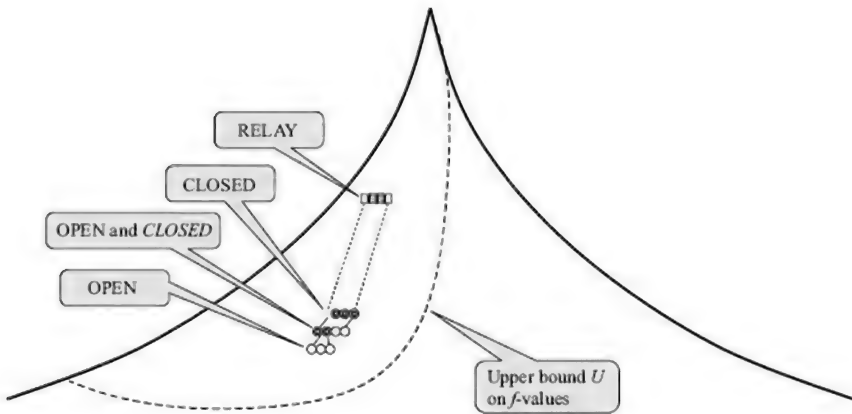


FIGURE 5.32 Divide and Conquer Beam Stack Search stores four layers of width w .

DCBSS differs from *DCBS*, in that for the sake of completeness it may backtrack and explore a different segment of the search space. Backtracking is however not a straightforward process because *DCBSS* has burnt its bridges and pruned away most layers from the beam. Instead of backtracking chronologically, the algorithm instead exploits the beam stack that maintains f_{\min} and f_{\max} values for all layers. It simply regenerates the layer it is required to backtrack to using the f_{\min} and f_{\max} values to guide the (forward) regeneration process.

One has to be careful though that one begins with a finite value of the upper bound U . Otherwise, it is possible that the algorithm may go into a loop, exploring the same nodes again and again on an undirected graph⁴ (see Exercise 5.11). This happens because the algorithm is not maintaining a complete boundary layer that would prevent search from leaking back to nodes seen earlier. However, selecting an appropriate U value will check this looping. The g -value of a node in the loop increases with each repetition. Once the f -value becomes greater than U , then search will break out of the loop.

Observe that the solution reconstruction with divide and conquer process will also require memory. One can opt to reconstruct the solution when backtracking is to be done. That is, the solution reconstruction is

delayed till an opportune moment. In the meantime, the algorithm can simply keep track of the best solution found (remember it continues to look for better solutions) and use its cost to update the upper bound to a tighter value.

Divide and Conquer Beam Stack Search is thus a complete and admissible search that has almost constant space requirement. The algorithm can explore deep search spaces without running into space problems. The larger the beam width w , the less likely is the need for backtracking, and one can thus choose a larger beam width w .

5.12 Discussion

In this chapter, we have explored algorithms for finding optimal cost solutions when the moves have an associated cost. Starting with the uninformed and conservative *Branch & Bound* algorithm (or *Dijkstra's Algorithm*), we added a heuristic function to exert a pull towards the goal on the search frontier. The resulting algorithm A^* is admissible when the heuristic function underestimates the distance to the goal. That is, the algorithm A^* is guaranteed to find the optimal solution, even if the heuristic function makes errors in judging which of the candidates is closer to the goal. Further, if the heuristic function is consistent, underestimating the cost of each move, the algorithm keeps finding optimal partial paths as it progresses.

The main drawback of A^* is that space requirement tends to grow rapidly with depth, because heuristic functions are rarely perfect. This limits the size of problems that can be tackled. IDA^* is an iteratively deepening algorithm that uses f -values to limit depth first search depth. *Recursive Best First Search* is an improvement that uses heuristic to guide search, but backtracks when necessary. Both the algorithms face problems on search spaces that are graphs because they find multiple paths to the same nodes. Recent problems from computational biology have thrown up such problems, spurring further research into the area. In particular, the problem of sequence alignment has posed new challenges. These problems were approached with algorithms that traded space for time. One strategy is pruning the *CLOSED* list, but at the expense of having to spend extra time reconstructing solutions found. The reconstruction procedure embodies a divide-and-conquer strategy, adopted from earlier work in dynamic programming.

For problems where the search space itself grows exponentially, the list *OPEN* is more of a problem. *Beam Search* prunes *OPEN* to operate with space growing only linearly with depth, but at the cost of completeness. The idea of a beam stack to do additional bookkeeping allows the *Beam Stack Search* to backtrack even after finding the first solution and look for a better one. This gives us an anytime algorithm that can be called upon to return a (good) solution on demand, along with the option to wait for the optimal solution.

Combining the two ideas of *Divide and Conquer* reconstruction and the *Beam Stack* systematic search, gives us an algorithm *DCBSS* with almost constant space requirements. Thus, one can tackle arbitrarily large search spaces.

All the above algorithms save on space as compared to A^* , but at the cost of extra running time. Every practical problem to be solved could use a variation that makes an appropriate trade off.

The algorithms seen so far view the problem solving process as a trajectory in some space, starting with a given situation and finding a path towards the solution. Each move transforms the entire representation, state or solution, to a different one. In the next chapter, we look at an approach in which a problem can be broken down into smaller problems, each of which can be attempted independently.



Exercises

- Figure 5.33 below shows the graph being explored by algorithm A^* at the point when node N is about to be expanded. The shaded double circles represent nodes on CLOSED, and the unshaded circles represent nodes on OPEN. Values inside the circles represent g -values of nodes. The two nodes marked *New* in dashed circles are about to be put on *OPEN*. The arrows depict back pointers and labels on edges denote costs.

Redraw the graph after A^* has finished expanding the node N .

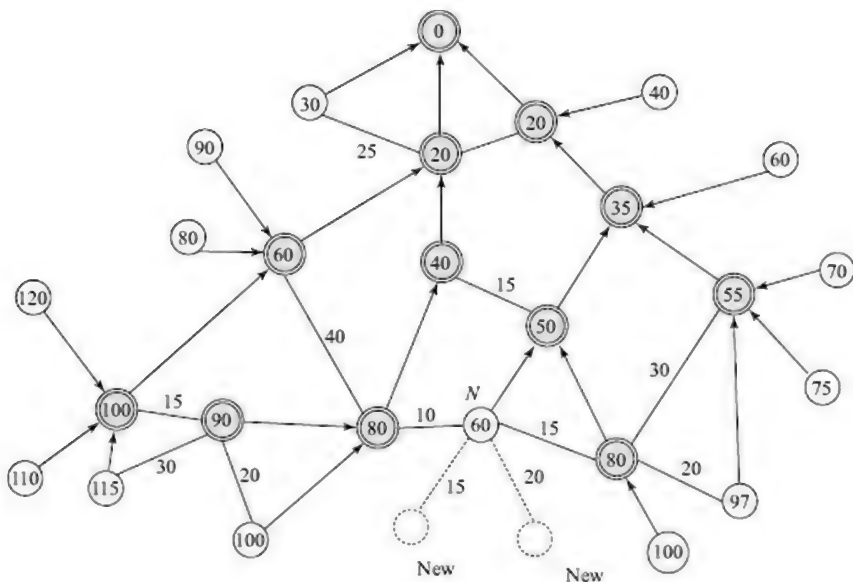


FIGURE 5.33 A* is about to expand node *N* in the above graph. How will the resulting graph look like?

2. Given the two graphs from Chapter 3 reproduced below, the task is to find an optimal path from the start node *S* to the goal node *G*. The length of each edge is marked on the graph. Use the Manhattan distance as the heuristic function. Assume that each unit on the grid is 10 kilometres.
 - (a) List the nodes that are on *OPEN* or *CLOSED* when A* terminates, and indicate their *f*-values.
 - (b) Show the order in which A* adds nodes to *CLOSED*.
 - (c) Mark the parent pointers on the graph as and when they are assigned. Show clearly if they are reassigned.

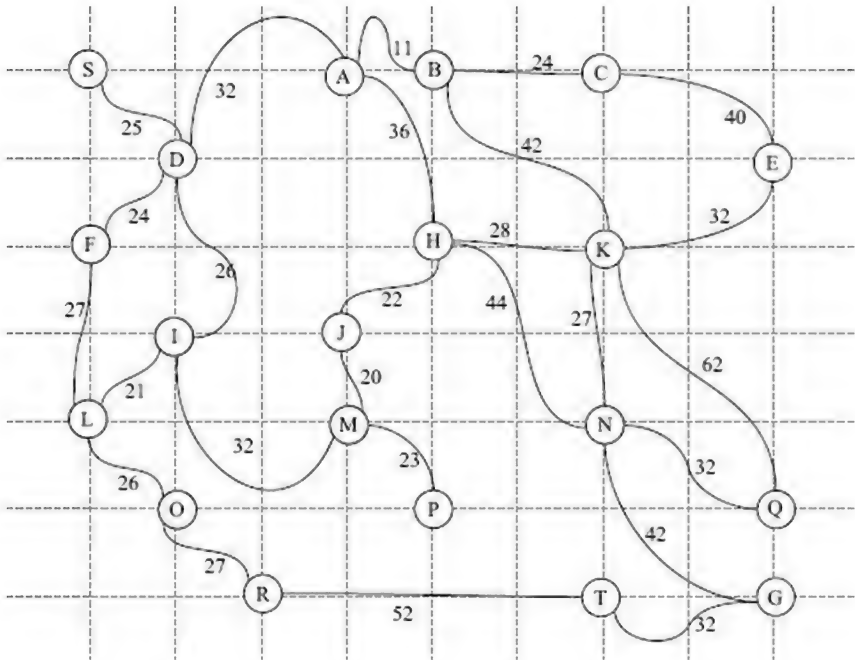


FIGURE 5.34 The graph from Figure 3.22 with weights assigned to edges.

3. Does the Manhattan-distance heuristic function satisfy the *monotone criterion* for the given problem? Would the Euclidean-distance heuristic function do so? Justify your answer.
4. Consider the state on the left for the 8-puzzle in Figure 5.36. Given that the goal state on the right, how does one define an accurate heuristic function? Consider the pairs of tiles {2, 3}, {4, 8} and {5, 6}. In each pair, the two tiles are in the correct row but in the *wrong order*. How would one design a heuristic function that accurately reflects the fact that extra moves have to be made to correct the order and move the tiles to their respective positions in the goal

state?

5. Replace the function $f(n) = g(n) + h(n)$ with the weighted function $f(n) = g(n) + K * h(n)$ for the 8-puzzle and 15-puzzle algorithms. Implement the algorithm and try out different values of K . Report the number of nodes examined for each value of K and the length of the solution found. Conduct the experiment for a set of randomly chosen problems.

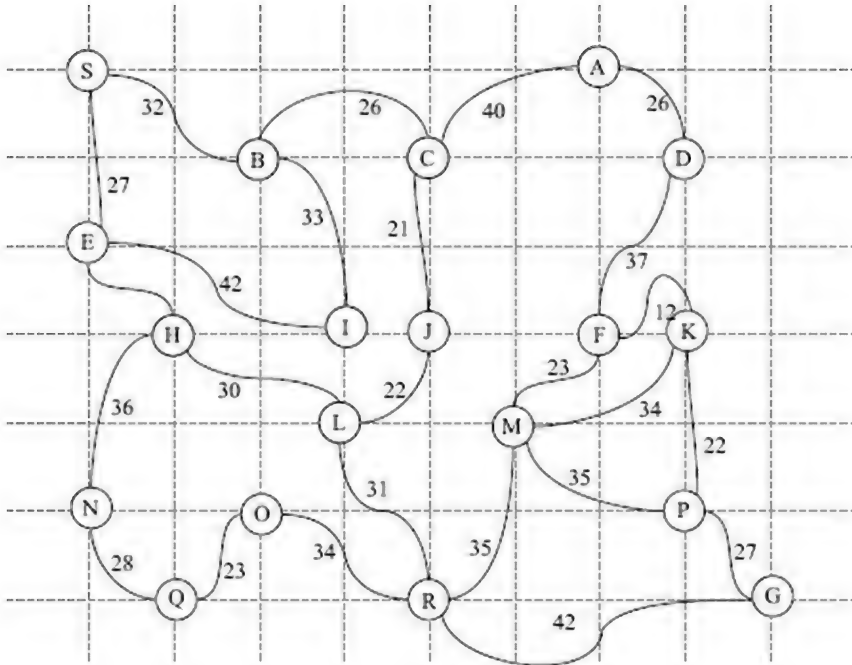


FIGURE 5.35 The graph from Figure 3.23 with weights assigned to edges.

6. Given the following three conditions, Arvind claims that the A^* algorithm may not terminate with the shortest path.
 - (a) Branching factor is finite.
 - (b) Cost of each edge is greater than zero.
 - (c) The heuristic function underestimates the distance to the goal.
 Do you agree or disagree? Justify your answer.



FIGURE 5.36 What should be the heuristic value of the state on the left, with respect to the goal state on the right?

7. Show that the node n being expanded by A^* in Section 5.6.1

- (Lemma 7) is in fact identical to node n_{L+1} .
8. Which of the following is more amenable to parallelization? *Best First Search*, *A* IDA**, *RBFS*. Justify your answer.
 9. Figure 5.37 shows part of a search space being explored by *RBFS*. The double circled nodes are on *CLOSED* and single circles are on *OPEN*. The dashed circled nodes and edges are yet to be generated. The values in the circles are *f*-values. Show how the graph will look after two more nodes are expanded. Mark the node that will be inspected next in the resulting graph.
 10. Implement a program to accept two strings from the vocabulary {C, A, G, T} and align the two strings. Assume the following costs. A mismatch costs 3 and a gap insertion costs 2 units. Two matching letters incur a cost of 0. Allow the user to change these costs and observe the alignments produced. Display the aligned sequences and the cost of the solution.
 11. Construct a map of your city or locality. Assign costs of edges based on the length of roads and the amount of traffic expected on that road. Implement the *Beam Stack Search* and the *A** algorithm for route finding. Compare the performance of the two algorithms in terms of nodes inspected and solution found. Try different beam widths.

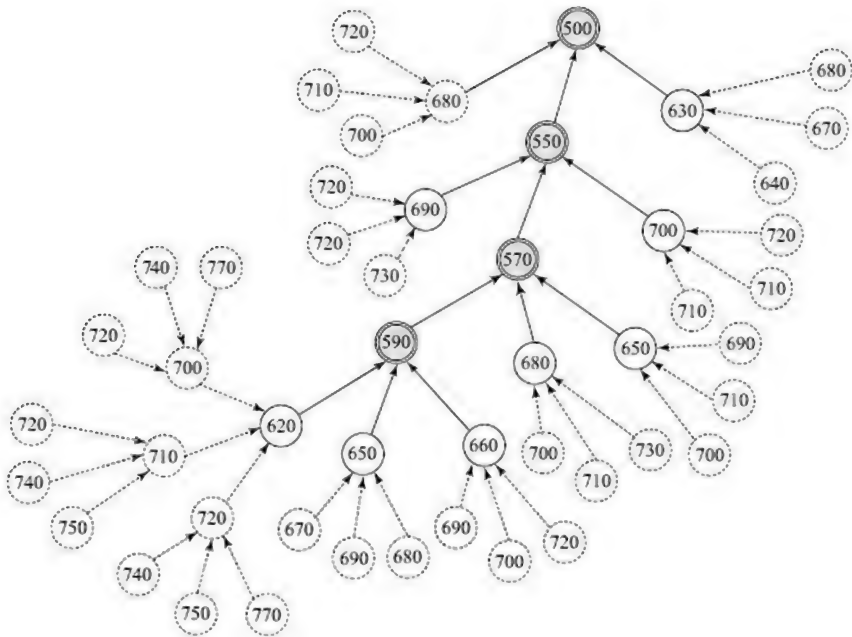


FIGURE 5.37 A graph being explored by RBFS. Numbers in the nodes denote *f*-values.

12. Simulate the DCBSS algorithm on the graph in Figure 5.37 with a beam width 2. What are the successors of nodes *F* and *G* in the layer

that follows? Assume that $h(n) = 0$ for all nodes. Try the algorithm with an admissible heuristic function (with approximate distances).

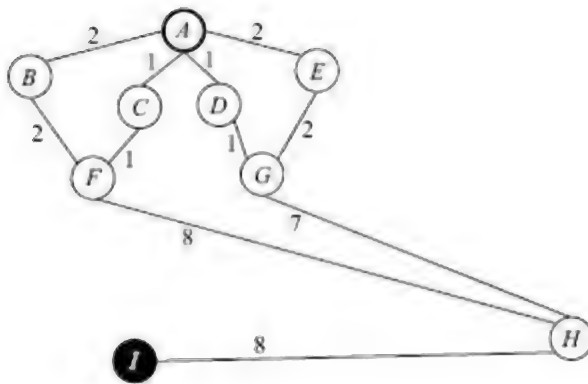


FIGURE 5.38 A small problem graph. Assume A is the start node, and I the goal node. Edge labels are costs associated with the moves.

- ¹ The goal may change with time. Ergonomics in kitchen design earlier had the goal of minimizing the movement of a person in a kitchen. Recommendations in today's sedentary world may be to increase the movement in order to provide exercise!
- ² One might think that it is enough to assume that the cost of each arc is positive. But if arc costs are real, then one could conjure up a problem where there exist paths of infinite steps, but whose total cost is smaller than a given value — first pointed out by Arvind Narayanan, a student, during my AI class at IIT Madras in the mid-nineties. See also Zeno's paradox in (Hofstadter, 1999).
- ³ Korf and his group describe it as being marked "used":
- ⁴ This observation was made by I. Murugeswari.

Problem Decomposition

Chapter 6

The problem solving approach we have explored so far is to search for a sequence of moves *starting* from the given state in the state space, or a candidate solution in the solution space, till a desired state or solution is reached. This problem solving strategy is essentially forward looking trial and error. In many situations, one can instead reason backwards from the goal to determine what needs to be done. That is, the search algorithm reasons in a backward fashion from the goal, rather than in the forward direction from the given. Of course, in some problems this may not make a difference, for example in path finding in a city map where the two approaches are equivalent. But in some problems, backward reasoning can lead to breaking up the problem into smaller parts that can be tackled independently, leading to smaller search spaces.

Consider, for example, the task of designing a treat for your friend. The “moves” could be choosing from different activities, and the goal could be an evening plan acceptable to your friend. Let us say that the evening plan constitutes three phases. You start with some activity, followed by a movie and dinner. Let the options be,

Evening → Visit Mall | Visit Beach

Movie → The Matrix | Artificial Intelligence: AI | Bhuvan Shome | Seven Samurai

Dinner → Pizza Hut | Saravana Bhavan

where the above *productions* represent the choices for each phase. Let us say that the forward search program traverses the tree shown in Figure 6.1 before terminating. Your friend is happy with a walk on the beach followed by the movie ‘The Matrix’, and dinner at Saravana Bhavan, as shown by the leaf node in grey in the figure. One can inspect the tree in some order, but observe that it is fruitless to search in the left subtree, which has *Visit Mall* as the activity. But a depth first search will end up doing exactly that, spending time searching the *entire* subtree below *Visit Mall*, before moving to the right half of the search tree. When it fails in one subtree below, it backtracks to the *last* choice made and tries the next subtree. That is, it does *chronological* backtracking.

One problem with chronological backtracking that *DFS* does is that it simply goes back to the last choice point and tries the next option. If after trying the first combination (Visit Mall, The Matrix, Pizza Hut), the algorithm somehow knew that the culprit for failure was the Visit-Mall

choice, it would backtrack directly to trying the next option at that level. We will visit this strategy, known as *dependency-directed backtracking* in Chapter 9. Meanwhile, let us focus on problem decomposition with backward reasoning.

The DFS search formulation above is a bottom-up approach, in which the algorithm synthesizes different combinations of primitive moves, and then tests for the goal being achieved. In this chapter, we explore an alternative approach in which we view problem solving as a top-down process. The main idea is that problems can be decomposed into subproblems. This decomposition process continues till we have problems that are trivial to solve. This could happen when we have a library of simple (primitive) problems and their solutions. This approach assumes that the problems can be decomposed into smaller problems that can be solved independently, in smaller search spaces.

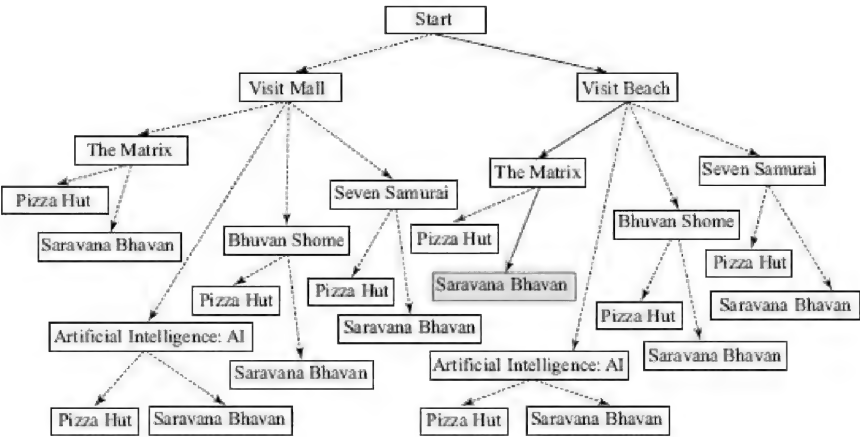


Figure 6.1 A search tree for planning an evening out. The desired plan is marked by the shaded node. Depth First Search will search through from left to right.

For the above problem of planning an evening out, we decompose the problem into three parts to be solved independently. They correspond to the three phases of the evening plan. The resulting search tree is depicted in Figure 6.2. Notice that the decomposition happens at the top level in this problem. This is indicated in the figure by connecting the three edges emanating from the root. In general, decomposition could also happen lower down as one breaks down a subproblem further.

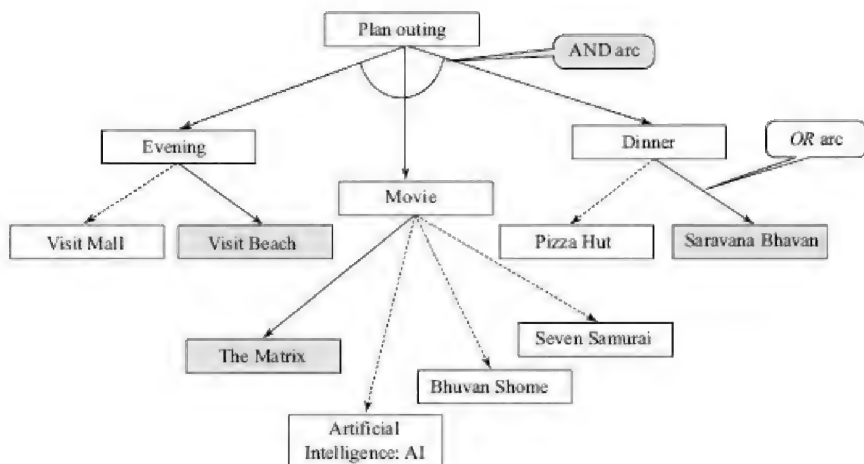


Figure 6.2 An AND-OR tree. And arcs represent subproblems to be solved individually. Notice that the solution is subtree rather than a path.

The semantics of an *AND* arc is that *all* the connecting edges have to be traversed. The three edges in the figure together can be thought of as a hyper-edge. Traditionally, such edges are known as *AND* edges, because all individual edges have to be traversed. In contrast, the other edges (the kind that we have been dealing with all along in the preceding chapters), are known as *OR* edges. This is because one could go down one edge, or the next one, or the next one. If a node has only an *AND* edge coming out of it, we will call it an *AND* node. Likewise, if it has only *OR* edges coming out, we will call it an *OR* node. In general, if we have a graph with both kinds of nodes, we can always convert them into graphs with pure *AND* and *OR* nodes by an addition of extra nodes, as illustrated in Figure 6.3. These search spaces are also known as *goal trees*, (Charniak and McDermott, 1985) because they break up a goal (problem) into subgoals (subproblems). We also call them *AND/OR trees/graphs* or *AO trees/graphs*.

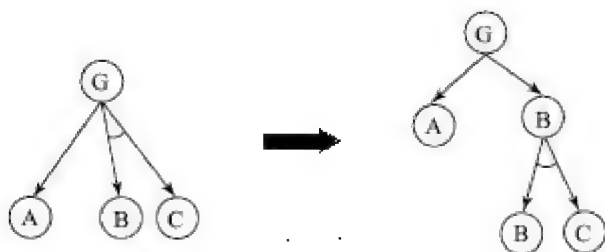


Figure 6.3 An AO graph with mixed nodes can be converted into a graph with pure AND and OR nodes.

Since one has to traverse all the edges at an *AND* node, the solution obtained will not be a path, but a subtree (or a subgraph) in the *AND/OR* search space. This is illustrated as solid arcs in Figure 6.2 for the *AND/OR*

problem formulated above. Observe that one can still use a depth first search approach, with the modification that at each *AND* node, more than one search will be spawned. In the above example, these will be three searches. The first will search below the node labelled *Outing*, the second below the node labelled *Movie*, and the third below the node *Dinner*. But since they will search independently, they will not explore fruitless combinations as done by our first formulation. In general, of course, an *AND/OR* search space will be much larger, with many *AND* and *OR* levels, and we will still need to adopt a heuristic approach.

We look at a couple of examples where *AND/OR* graphs have been used for building problem solvers.

6.1 SAINT

One of the first AI systems that used AO graphs was *SAINT*, developed as part of his doctoral thesis by James Slagle (1961). *SAINT* was designed to solve symbolic problems in mathematics (Slagle, 1963), and was a precursor to many subsequent systems. We look at an example where *SAINT* solves an integral equation by searching for transformations and problem decomposition. The given problem is,

$$\int \frac{x^4}{(1-x^2)^{5/2}} dx$$

Through a series of transformations as shown in Figure 6.4, *SAINT* breaks it down to three simple problems that can be solved trivially. The reverse transformations are applied to the three solutions to finally get the solution,

$$\frac{1}{3} \tan^3(\arcsin x) - \tan(\arcsin x) + \arcsin x$$

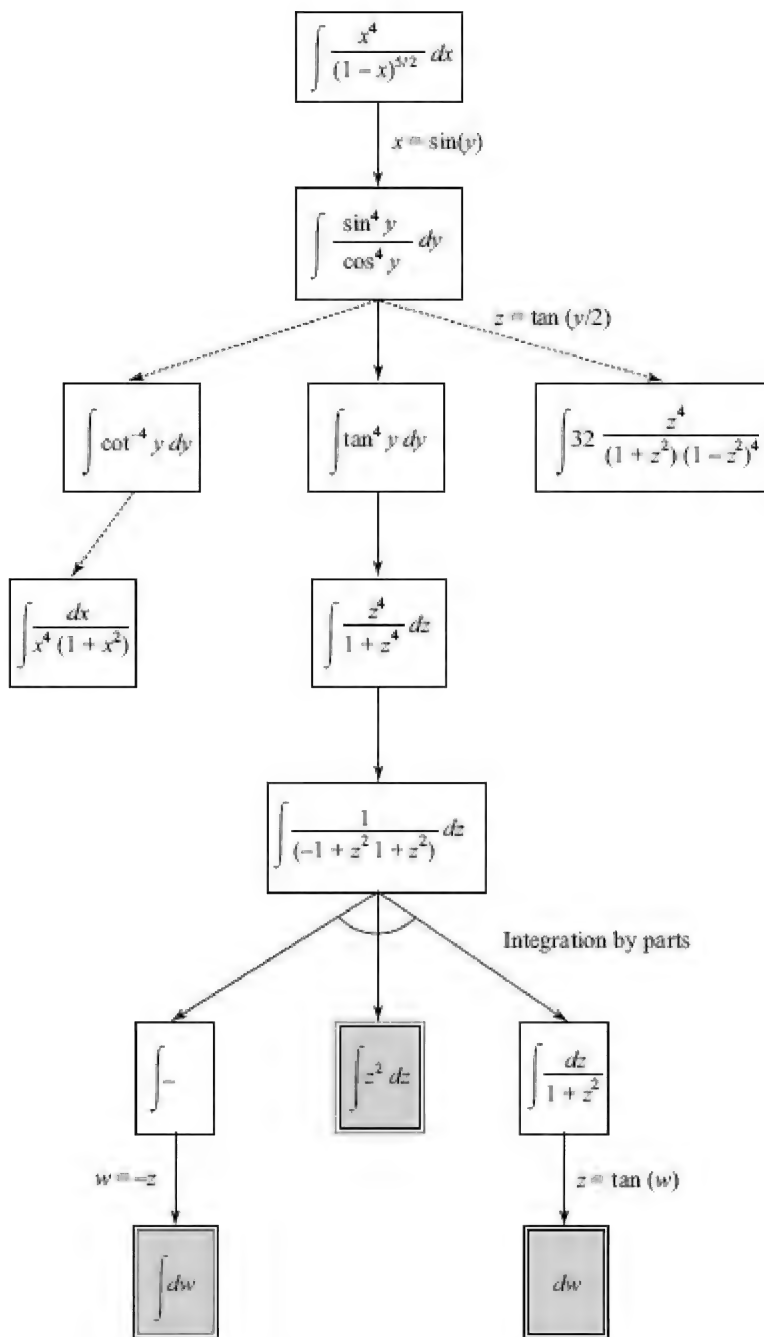


Figure 6.4 Symbolic integration as an AND-OR search problem (Nilsson, 1971).

6.2 Dendral

One of the earliest successes of AI was the program *Dendral* (for Dendritic Algorithm), developed at Stanford University during 1965–1980, by a team led by Joshua Lederberg, Edward Feigenbaum, Bruce Buchanan and Carl Djerassi (Lindsay et al., 1980, 1993). The *Dendral* program was the first AI program to emphasize the power of specialized knowledge over generalized problem-solving methods.

The objective of the program was to assist chemists in the task of determining the structure of a chemical compound. This problem is important because the chemical and physical properties of compounds are determined not just by what their constituent atoms are, but by the arrangement of these atoms as well¹. The difficulty is that the number of candidate structures (called hypotheses) for a given compound can be very large, and grows combinatorially (running into millions) as we consider molecules with more and more atoms. *Dendral* led to a program called *CONGEN* (CONstrained GENerator) that allows a chemist to constrain the generation of candidates. Figure 6.5 shows some of the hypotheses generated by *CONGEN* for the compound $C_6H_{13}NO_2$.

The candidates generated were used as inputs to a system that produced synthetic spectrograms which were then tested against real spectrograms of the given compounds, before presenting them to the user. The key to its success was to use its knowledge of chemistry in the form of rules about allowed and taboo connections, to consider only plausible candidates. The success of the program is illustrated by the following quote:

“By observing structural constraints within molecules which made certain combinations of atoms implausible, generating and testing hypotheses about the identity of the compound, and ruling out candidates that did not fit within the structural constraints, Dendral traced branches of a tree chart that contained all possible configurations of atoms, until it reached the configuration that matched the instrument data most closely. Hence, its name, from “dendron, “the Greek word for tree. ... In its practical utilization, Dendral was designed to relieve chemists of a task that was demanding, repetitive, and time-consuming: surveying a large number of molecular structures, to find those that corresponded to instrument data. Once fully operational, the program performed this task with greater speed than an expert spectrometrists, and with comparable accuracy.”²

Figure 6.6 illustrates the kind of search space explored by *Dendral*. The reader will recognize it as an AO graph.

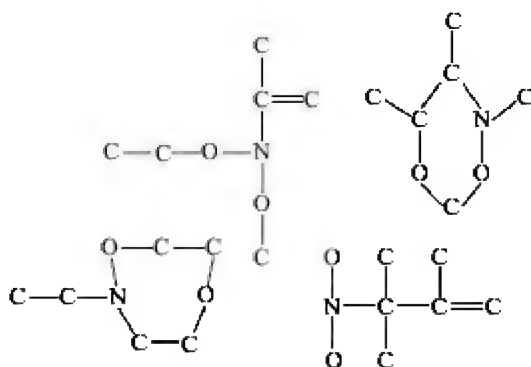


Figure 6.5 Some of the different candidate structures for $C_6H_{13}NO_2$ generated by *CONGEN* (Buchanan, 82). The hydrogen atoms are not shown. They can be filled up, based on valence.

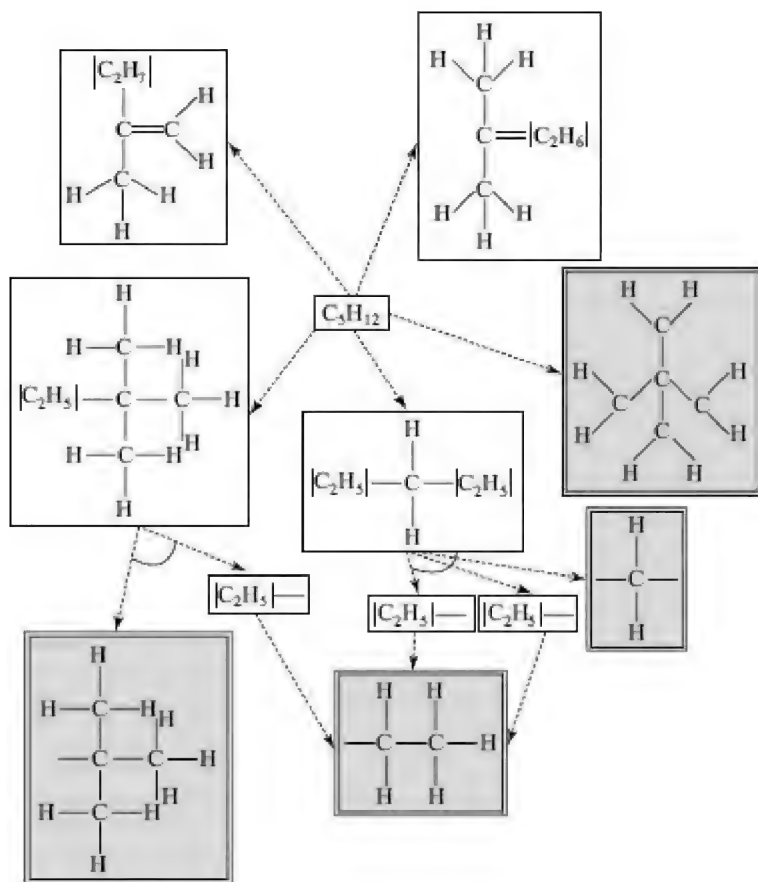


Figure 6.6 The search space for *Dendral* is an AO graph (figure adapted from (Mainzer, 2003) Chapter 6). The double line nodes are SOLVED nodes.

6.3 Goal Trees

We look at a heuristic algorithm to search goal trees. We assume that we have a heuristic function that estimates the cost of solving each node.

The solution of an AO problem involves reduction to a set of primitive problems which have trivial solutions. We will label the primitive problems in the AO graph as *SOLVED*, to indicate that no further reduction is required. We will assume that *SOLVED* nodes have a certain cost associated with them. This cost may be zero in problems like symbolic integration, where the solution is directly available. This cost may also be nonzero for some problems. For example, if we were to pose the problem of constructing a house then one of the primitives of our system could be “install a door”, which would have a cost associated with it. We may also have costs associated with each arc or edge, representing the problem transformation cost. Whatever the costs associated with a problem, we will assume that a heuristic function estimates the total cost of *solving* a given node. Furthermore, we will require the heuristic function to be a lower bound on the actual cost of solving a node, in order to ensure that the optimal cost solution is found. The argument for underestimating the actual cost is similar to the one presented for *Branch&Bound* and *A** algorithms seen in Chapter 4. That is, as long as there is the possibility of finding a cheaper solution, the algorithm will continue searching even after one has been found.

Before writing the algorithm for solving goal trees, let us cook up a small problem and investigate how search could progress, using a heuristic function. Let us say that we start with a goal *G* of estimated cost 45, and expand it to get two ways of solving it, one an *OR* arc and the other an *AND* arc as shown in Figure 6.7. Let the three successor nodes *A*, *B* and *C* have heuristic values as 42, 22 and 24 respectively. Let the three edges leading to them have costs 4, 3 and 2 respectively. Which node should the algorithm refine (expand) next?

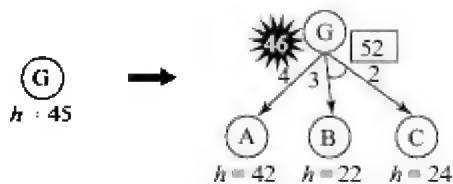


Figure 6.7 Which is the best node to expand next? Even though node *B* has the lowest heuristic value it is a part of a more expensive looking option. This is because it is a leaf on an *AND* (hyper) arc. The estimated cost of that solution is $22 + 3 + 24 + 3 = 52$. The more expensive looking node *A* has an estimated cost of $42 + 4 = 46$. Note that the choice is made on the basis of backed up values 46 and 52.

Unlike the *A** search for optimal solutions, (paths) a node in the AO graph is not a representative of a solution. This is because solutions are not usually paths but subtrees, and a node may have an *AND* sibling (or a cousin) which also contributes to the solution with its own associated cost.

Thus, looking at the heuristic value of a node by itself may not be useful. Even counting the costs of the edges leading to the node (like the g -values in A^*) will not help, because the node only accounts for a part of the solution. Instead, we need to look at the total estimated cost of a solution. In the above example, the total estimated cost of the two options is 46 and 52 respectively. After every expansion by the algorithm, the best choice at each node is marked. The search should refine the marked solution. In the above example, this means that it should expand the node A next.

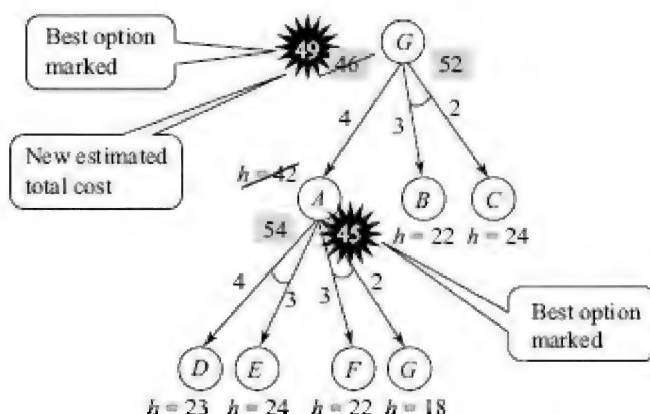


Figure 6.8 After node A is expanded, there are two options (D , E) and (F , G) with estimated costs 54 and 45 respectively. The revised estimate of A now becomes 45, the lower of the two. Propagated back to root, this option has a revised cost of 49. It is still a better option, and AO^* will now go down the marked path and expand one of F and G .

The resulting graph is shown in Figure 6.8 and it represents a typical AO graph, not yet solved completely. Observe that there are two choice points in the graph, and at both, the best solution is marked. The topmost task at the root has two options. Let us call them the left option and the right option. In general, of course, there may be any number of options. The left option has been refined and it itself has two options for further decomposition. Again, the best option is marked. The left option, represented by node A , had a heuristic estimate of 42 before expansion, which was revised to 45, the better of the two sub-options, represented by nodes F and G , and marked as the better option. The revised estimate of node A has to be propagated up, and the left option at the root now evaluates to 49 instead of 46.

In every cycle of refinement, the algorithm starts at the root. It follows the marked best option at each choice point, until it reaches some yet unsolved node or a set of nodes. It will refine one of them, and then propagate the new values back up again. In the process if the subproblems of a given node are *SOLVED* nodes then it may also propagate *SOLVED* label back. The algorithm will terminate when the *SOLVED* label is backed up right up to the root.

Thus, the algorithm for solving the goal tree, known as the AO^* algorithm (Martelli and Montanari, 1978; Nilsson, 1980), has the following

cycle:

- Starting at the root, traverse the graph along marked paths till the algorithm reaches a set of unsolved nodes U .
- Pick a node n from U and refine it.
- Propagate the revised estimate of n up via all ancestors.
- If for a node all *AND* successors along the marked path are marked *SOLVED*, mark it *SOLVED* as well.
- If a node has *OR* edges emanating from it, and the cheapest successor is marked *SOLVED* then mark the node *SOLVED*.
- Terminate when the root node is marked *SOLVED*.

The detailed algorithm given in Figure 6.9 below has been adopted from (Rich and Knight, 1991).

6.3.1 An Example Trace of AO*

Let us look at a complete example depicting the progress of the AO* algorithm on a synthetic problem. We begin with a version of a problem where the heuristic function is *not* a lower bound on the actual cost of solving each node. In the following example (Figure 6.10), assume that the *SOLVED* nodes all have an associated cost zero. The labels on the nodes are heuristic values. Let the cost of every arc in the graph be one. This means that the cost of solving a node is dependent only on the number of arcs leading from it to *SOLVED* nodes. A cursory glance at the figure reveals that the heuristic function shown in the figure is quite wild. It definitely overestimates the cost of solving nodes specially the ones closer to *SOLVED* nodes.

Figures (6.11 continued in 6.12) show the progress of the AO* algorithm. At each stage, the algorithm goes down the marked path and expands the node shown in bold. The backed-up values in the nodes are the best known cost estimates for that node, backed up after each expansion.

After the last node in the above sequence is refined, the algorithm terminates with a solution as shown in Figure 6.12.

The cost of the solution found is 8, but the heuristic functions in many places had much higher estimates. An overestimating heuristic function makes the search opinionated. It refuses to consider unseen alternatives because it estimates them to be worse. In the process, it may miss better solutions. For example, the AO graph has a better solution costing 7 units which the algorithm misses.

```

Algorithm AO*()
1  /* uses a graph G instead of open and closed
2   the graph G is initialized to the start node start */
3  G ← start
4  Compute h(start)
5  while start is not labeled SOLVED AND h(start) ≤ Futility
6  do      /* Futility is the maximum cost solution acceptable */
7      /* Forward phase */
8      Trace the marked path leading to a set U of unexpanded nodes
9      Select node n from U
10     children ← successors of n
11     if children is EMPTY
12         then h(n) ← Futility
13     else Check for looping in members of children
14         Remove any looping members
15         for each s ∈ children
16             do Add s to G
17                 if s is primitive
18                     then Label s SOLVED
19                         Compute h(s)          /* could be zero */
20     /*Propagate Back*/
21     /* Let M be the set of nodes that are modified */
22     M ← n
23     while M is not EMPTY
24         do Select deepest node d from M, and remove it from M
25             Compute best cost of d from its children
26             Mark best option at d as MARKED
27             if all nodes connected through marked arc are labeled SOLVED
28                 then Label d as SOLVED
29             if d has changed
30                 then Add all parents of d to M
31 if start is marked SOLVED
32     then return marked subgraph starting at start
33 else return FAILURE

```

Figure 6.9 Algorithm AO*.

The way to ensure that the optimal solution is found is by making sure the heuristic function underestimates the cost of solving a node. In this problem, this can be done by dividing the heuristic value by ten. Equivalently, we can assume each arc to cost 10 units, which makes the heuristic values given in problems as underestimating the actual costs. The progress of the algorithm is shown in Figures 6.13 and 6.14.

The underestimating function does not grab the first solution in sight, and continues searching till no better options are left. The progress of the algorithm is continued below.

The algorithm terminates with the optimal solution costing 70 shown in the right in Figure 6.15. In fact, at that point, it has also discovered another optimal solution shown in the figure on the left.

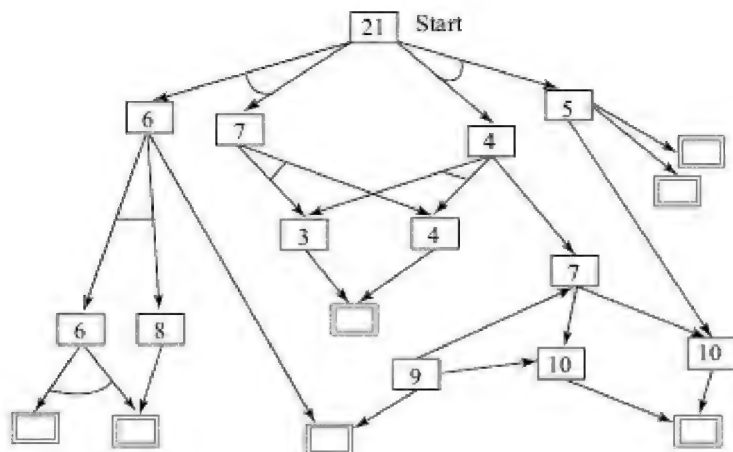


Figure 6.10 A synthetic *AND/OR* problem. Assume every arc costs one unit to traverse. Nodes are labelled with heuristic values. Solved nodes represented by double-lined boxes have cost zero.

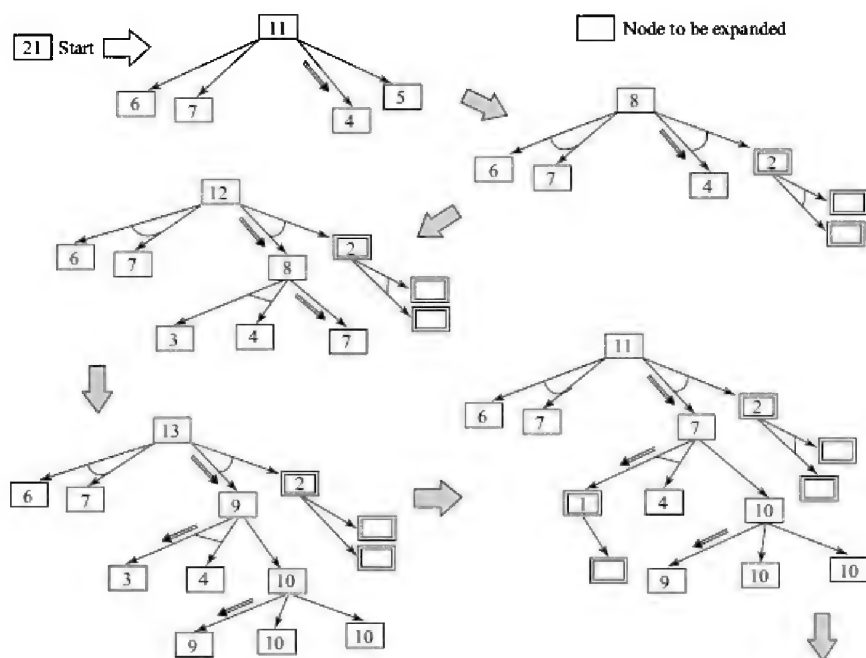


Figure 6.11 The progress of the AO* algorithm. The best options at each choice point are marked by arrows. Nodes in double-line squares are *SOLVED*.

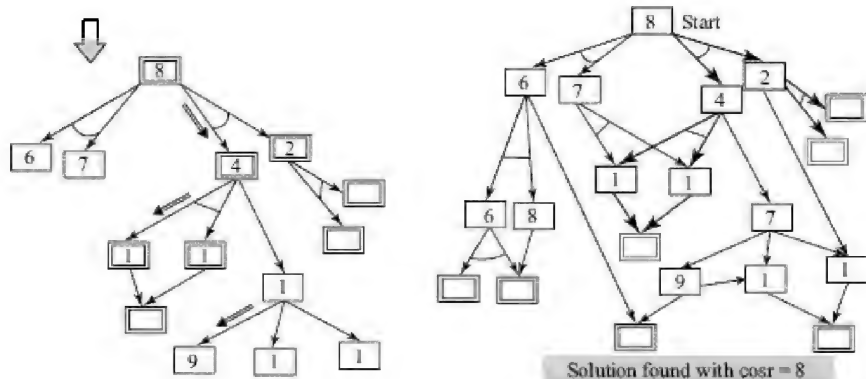


Figure 6.12 The algorithm terminates as the root (start) is labelled *SOLVED* (double square). The solution is shown in bold lines on the right.

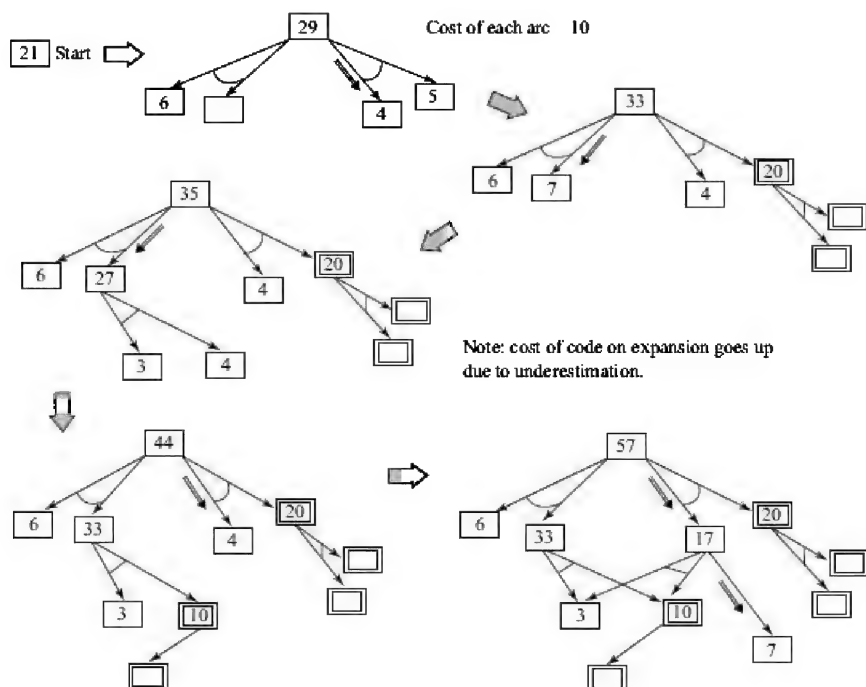


Figure 6.13 Searching with an underestimating function. Observe that an underestimating heuristic function makes the search try out more options at each stage. Searching with an overestimating function had plunged down one path.

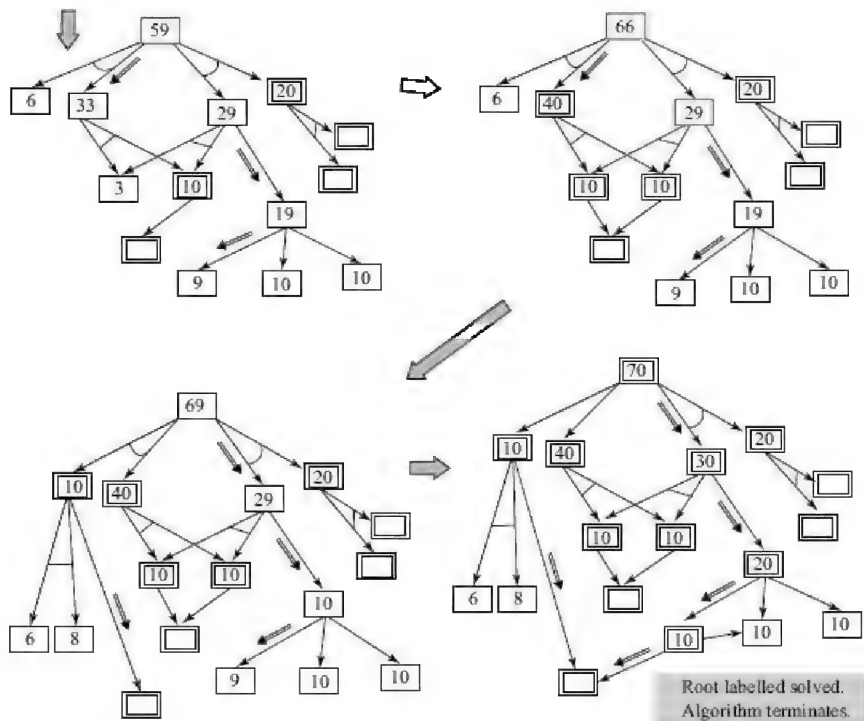


Figure 6.14 AO* terminates after exploring a larger search space, but finds an optimal solution.

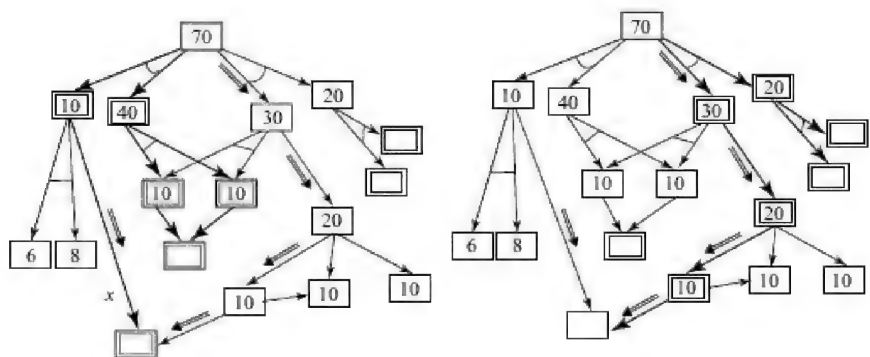


Figure 6.15 AO* terminates with the optimal solution on the right with cost 70. It has also found the solution on the left at termination. If the arc marked *x* were to be 9 instead of ten, it would have reported that solution with cost 69.

6.4 Rule Based Systems

Rule based systems or *production systems* have been used in general to decompose a problem and address it in parts. In its most abstract form, a rule or a production is a statement of the form,

Left-Hand Side → Right-Hand Side

in which the computation flows from the left-hand side to the right-hand side. In the example we saw in the beginning of this chapter, productions were used to capture goal-directed knowledge about the domain. The production,

Dinner → Pizza Hut | Saravana Bhavan

says that the goal of having dinner can be reduced either to the goal of eating at Saravana Bhavan or eating at Pizza Hut. The production system was used to break down a given goal into a set of primitive goals that could be solved trivially.

Productions or rules can also be used in a forward direction, in a *data-driven* manner. The production then looks like,

Pattern → Action

where the pattern is in the given database. Thus, a rule based system looks at a part of a state, and triggers some action when a pattern is matched. Usually, the actions are to make some changes in the database describing the state. That is, one in fact achieves a state transition but only by dealing with a part of a state that is characterized by the pattern.

For example, one could write a rule to sort an array of numbers as follows:

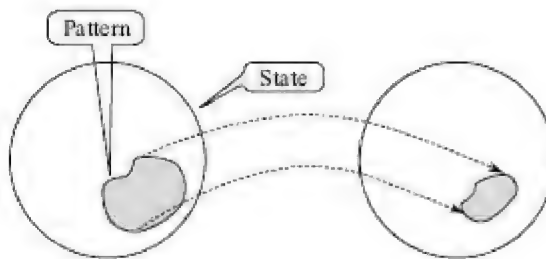


Figure 6.16 A rule looks at a part of a state and modifies it to result in a new.

```
{p interchange
  {array ^ index i ^ value N}
  {array ^ index {j > i} ^ value {M < N}}
  →
  {modify 1 ^value M}
  {modify 2 ^value N}}
```

We have used the notation of the language OPS5³ (Forgy, 1981), one of the first rule based languages developed at Carnegie Mellon University. The above rule has a name “interchange” and reads as follows:

```

{rule interchange
  IF there is an element at index i with value N,
  AND IF there is an element at index j > i with value M < N
  THEN
    modify array(i) to hold M,
    AND modify array(j) to hold N)

```

Given an array of numbers, this rule will *keep firing* as long as it sees two numbers that are in an incorrect order, and will only stop when all pairs of elements are in order. While this rule will indeed sort a given array, one cannot make any statement about how efficient it will be in doing so. This is because we do not know which two elements it will look at any given time point. That would depend upon the conflict-resolution strategy adopted by the inference engine (describe below).

This rule gives us an insight into a strong point of rule based systems that the same rule can apply to different sets of data doing the same task with each one of them. Identifying what data a rule matches is done at runtime. Thus, based on the kind of processing we want to do, we could write a set of rules that will apply to patterns in any given state. This would imply that we may not have to write *MovGen* functions for doing state transformation. Instead, the state transformation (or moves) can be composed from rules. We will look at this approach in more detail in Chapter 7 on planning, in which we will use planning operators, which are like production rules, to construct sets of moves to search over.

Both the examples above depict rules being used with *forward chaining*. That is, the left-hand side is used to match the rule and the right-hand side, the effect of using the rule. The other direction is also possible. The right-hand side can be used as a match pattern and the left-hand side defines the effect of using the rule. Such a strategy is called *backward chaining*.

However, the two examples differ in another aspect.

The example of production that reduces “dinner” to “Pizza Hut” or “Saravana Bhavan” is operating in a *goal-directed* fashion. It is saying that if the goal is to have dinner then it could be achieved by going to one of the two restaurants. This form of reasoning is also called *backward reasoning* because the reasoning is done backwards from the goal.

The second example, that is in fact a one rule program for sorting an array of numbers, is an example of *data-driven reasoning* or *forward reasoning*. This is because the rules are triggered by patterns in the data. The name *pattern-directed inference systems* has also been used to refer to such systems.

The above two rules illustrate the fact that both forward reasoning and backward reasoning can be *implemented* using forward chaining. In fact, all four combinations of forward and backward reasoning and forward and backward chaining are possible. In Chapter 12, we look at deduction in logic, backward reasoning with backward chaining, and also forward reasoning with forward chaining.

In the rest of this chapter, we explore how forward chaining systems

have been implemented to provide a platform for encoding problem solving knowledge in the form of rules obtained by domain experts. The idea took off in the seventies in the last century. The aim was to get experts in different domains to articulate their knowledge about the domain in the form of rules. Domain experts were not required to be competent in programming in order to build systems based on their knowledge. They needed only to express their knowledge in the <pattern, inference> form, and an *inference engine* would operationalize this knowledge in problem solving activity. Thus, the idea of *Expert Systems* was born.

We look at one system *XCON* briefly before delving into the algorithms that go into implementing the inference engine, also called an *expert system shell*.

6.5 XCON

Originally called R1⁴, the *XCON* system was a forward-chaining rule-based system to help automatically configure computer systems (McDermott, 1980a, 1980b). *XCON* (for eXpert CONFIGurer) was built for the computer company Digital Equipment Corporation, and helped choose components for their VAX⁵ machines. *XCON* was implemented in the rule based language *OPS5* described below. By 1986, *XCON* had been used successfully at DEC processing over 80,000 orders with an accuracy over 95%.

XCON is a forward-chaining rule based system that worked from requirements towards configurations, without backtracking. It needed two kinds of knowledge (Jackson, 1986):

- Knowledge about components, for example voltage, amperage, pinning-type and number of ports, and
- Knowledge about constraints, that is, rules for forming partial configurations of equipment and then extending them successfully.

It stored the component knowledge in a separate database, and used its production system architecture to reason about the configuration. The following is an example of a record that describes a disk controller.

```
RK611*  
CLASS:          UNIBUS MODULE  
TYPE:           DISK DRIVE  
SUPPORTER:      YES  
PRIORITY LEVEL: BUFFERED NFR  
TRANSFER RATE:  212 ...
```

Constraints knowledge is specified in the form of rules. The LHS describes patterns in partial configurations that can be extended, and the RHS did those extensions. The following is an English translation of an *XCON* rule taken from (Jackson, 1986).

DISTRIBUTE-MB-DEVICES-3

```
IF      the most current active context is distributing massbus devices
      & there is a single port disk drive that has not been assigned to a
      massbus
      & there is no unassigned dual port disk drives
      & the number of devices that each massbus should support is known
      & there is a massbus that has been assigned at least one disk drive
      and that should support additional disk drives
      & the type of cable needed to connect the disk drive to the previous
      device on the disk drive is known
THEN
      assign the disk drive to the massbus
```

Figure 6.17 An English translation of an *XCON* rule (Jackson, 1986).

The *XCON* rules were written grouped into rule sets catering to distinct subtasks in the configuration tasks, by specifically stating the context (see first condition in the above rule) to demarcate the different sets of rules. This makes the task of the underlying processing algorithm easier, because it needs to only look at a subset of rules at each stage. Further, *XCON* breaks up the subtasks themselves into smaller subtasks, resulting in a task hierarchy of the kind depicted in Figure 6.2. Thus, we see that with the introduction of *contexts* (or subgoals) as data elements, one can bring in a backward reasoning flavour even in a forward-chaining system.

The high-level task break up in *XCON* is as follows (Jackson, 1986).

1. Check the order, inserting any omissions and correcting any mistakes.
2. Configure the CPU, arranging components in the CPU and the CPU expansion cabinets.
3. Configure the unibus modules, putting boxes into expansion cabinets and then putting the modules into the boxes.
4. Configure the panelling, assigning panels to cabinets and associating panels with unibus modules and the devices they serve.
5. Generate a floor plan, grouping components that must be closer together and then laying the devices out in the right order.
6. Do the cabling, specifying what cables are to be used to connect devices and then determining the distances between pairs of components.

Thus, *XCON* fills in the low-level details of the above “algorithm” by selecting and firing specific rules. And since the entire implementation is in *OPS5*, the sequencing of the high-level tasks is controlled by activating the relevant contexts in the given order, somewhat like passing a baton in a relay race. McDermott said that the justification of such an approach of creating and working within specific contexts was that this was how the human experts approached the task.

6.6 Rule Based Expert Systems

The idea of *expert systems* that caught on, following the development of some of the systems described above, was that domain experts could

transfer their knowledge to systems in the form of *rules*, and a general purpose *inference engine* would string together rule firings to some problems using this knowledge. It led to the development of programming languages like *OPS5* (see also (Brownston et al., 1985)) which facilitated the expression of such rules.

Traditionally, an expert system is a rule based system and has the following components.

1. A Set or Rules The set of rules are like the long-term memory (LTM) of a problem solver (Newell, 1973). Rules are basically a memory of associations between patterns and actions. When some elements of data in the current problem match a pattern in a rule then that rule is ready to *fire* or *execute*.

2. A Working Memory (WM) The Working Memory of a problem solver is like its Short-Term Memory (STM). It stores the data related to the current problem. The working memory is made up of *Working Memory Elements* (*WMEs*). Each *WME* carries a time stamp marking when it was created, usually implicitly in sequence numbers assigned to them.

3. The Inference Engine The inference engine is a constant feature in all systems. It is domain independent, and drives the processing. The inference engine operates in the cycle described in Section 6.6.3, in which it picks rules with matching data and executes them.

6.6.1 Working Memory

The working memory is made up of an ordered sequence of *WMEs*. Each *WME* is an instance of a *WME class*. A *WME class* has the following structure,

```
{classname
  attribute-name-1
  attribute-name-2
  .
  .
  attribute-name-k}
```

For example, devising a bridge playing program, one might declare a class in *OPS5* by the following statement, which declares a class and its attributes.

```
(literalize card-rank
suit          ; spades, hearts, diamonds or clubs or s, h, d, c
card          ; a, k, ..., 3, 2
rank          ; 1, 2, ..., 12, 13
int-rank      ; internal rank
will-win      ; winner number from top
```

tempo	; losers above this card
player	; north ⁶ , south, east or west or N, S, E or W
points	; 0, 1, 2, 3, 4
rankclass	; rank in terms of trick-taking value.
	; for example a king supported by an ace has rankclass 1
played	; yes, no
updated)	; yes, no, marked, ranked

The text after the semicolon is a comment in *OPS5*. The class *card-rank* will have 52 *instances* in a card pack. The attributes are meant to keep track of the name, the current rank, the player holding it and whether it has been played or not amongst other features that are modified as the game proceeds.

A *WME* is an instance of a class, and has the following form:

```
{classname
  attribute-name-1
  attribute-name-2
  .
  .
  attribute-name-k}
```

That is, the *WME* has values in the various attribute slots. Observe that the index m used above is different from k in the class definition. In fact, m is less than or equal to k . This is because one does not have to specify values for all attributes in a *WME*. Furthermore, the order of specifying attributes in a *WME* is unimportant. Specifying attributes in arbitrary order does not increase the complexity of matching process, because of the compiled network that the *OPS5* system uses. The following are then valid instances of the class *card-rank*. Observe that white space is ignored and indentation is mainly for human consumption, and order of mentioning attributes and their values does not matter.

```
(card-rank ^card 9 ^suit spades ^player north ^played no)
(card-rank ^player south ^card a ^suit spades ^rank 1)
```

6.6.2 Rules

We have already seen some rules defined above. In *OPS5*, a rule has the following structure:

```

(p ruleName
  pattern-1
  pattern-2
  .
  .
  pattern-p
→
  action-1
  action-2
  .
  .
  action-r)

```

where each pattern is equal to or a generalization of a matching *WME*, optionally preceded by a negation sign “-”. However, the first pattern cannot have the negation sign. The pattern may itself be enclosed in a pair of angle brackets, along with a name assigned to the pattern.

The following patterns are discussed in the context of the two example *WMEs* above.

(card ^suit spades) and (cards ^suit <x>)

Both patterns will match both *WMEs*. The value <x> in the second pattern is a variable that can match anything. However, if the variable occurs in more than one pattern within a rule then it should match the same value,

(card ^suit spades ^player <<south north>>)

which will also match both *WMEs*. The value for attribute ^player <<south north>> is a disjunction. This means that the value could be either of *south* and *north*. The pattern can also be specified as a conjunction, as follows,

(card ^player south ^rank {<n> > 0 < 5})

This says that the value of rank of the card is called <n>, and that it should both be greater than 0 and smaller than 5. This will match the second *WME*, as will the following:

(card ^player south ^rank {<n> = 1})
 (card ^player south ^rank < 2)

Given a pattern *p* and a *WME* *w*, the match between the two is positive, if for every attribute of the pattern there is a matching attribute in the *WME*. By matching we mean that the value of the attribute in the *WME* satisfies the constraints expressed in the corresponding slot in the pattern.

The negation sign “-” inverts the match criteria. A pattern $\neg p$ in a rule is satisfied if there is *no* *WME* in the WM matching that pattern. Thus, for example, the following rule will match the highest ranking card held for the player *south* in each of the suits that has not been played,


```
(p top-card
  (card-rank ^suit <s> ^player south ^rank <r> ^card <c> ^played no)
  -(card-rank ^suit <s> ^player south ^rank <r> ^played no)
→
  (make (highest-card ^player south ^suit <s> ^card <c> ^rank <r>)))
```

The above rule will fire four times at the beginning of the game, once for each suit, and create new *WMEs*, specifying the highest card of *south* in each suit. If we had instead stated the value of *^player* as a variable *<p>* instead of the atom *south*, the rule would have fired for each of the four players.

Observe that we have also introduced one RHS action in this example. The action is

```
(make WME ...)
```

This has the effect of creating the specified *WME*. The *make* action can also be specified outside a rule, independently, to the *OPS5* interpreter, either directly from the keyboard, or in a file presumably containing a set of initializing actions.

Another action on the RHS is (*modify WME ...*) that modifies a *WME* identified in the LHS. The patterns in the LHS of a rule can be referred to, either by their sequence number, or an explicitly specified name. For example, we might have a rule that does some actions once a card is to be played expressed as followed.

```
(p update-top-card
  (played ^card <c> ^player <p> ^suit <s>)
  (card-rank ^card <c> ^player <p> ^suit <s>)
  (highest-card ^card <c> ^player <p> ^suit <s>)
→
  (remove 3)
  (modify 2 ^played yes))
```

Notice that the patterns on the LHS just need be enough to identify the concerned *WME* and access its relevant attributes. The pattern (*card-rank ...*) above is used to spot the relevant card and *modify* its attribute *played* to value *yes*, and one needs only enough information to pinpoint the *WME*.

Two actions have been introduced above. The first one says that the *WME* matching the pattern 3 of this matching instance of the rule should be removed or deleted. The second one says that in the corresponding matching *WME* for pattern 2, the value of the “played” attribute is to be set to “yes”. This will, of course, overwrite the earlier value of “no” before the card was played.

The above rule could equivalently be written as,

```

(p update-top-card
  (played ^card <c> ^player <p> ^suit <s>)
  {<pc> (card-rank ^card <c> ^player <p> ^suit <s>)})
  {<hc> (highest-card ^card <c> ^player <p> ^suit <s>)})
→
  (remove <pc>)
  (modify <hc> ^played yes))

```

This uses names for patterns instead of their sequence number.

Apart from the three actions—*make*, *modify* and *remove*—introduced here, the language also has actions for doing arithmetic computations (*compute ...*) and input output actions (*openfile ...*) (*closefile ...*), (*write ...*), (*accept ...*), (*crlf*), etc. The reader is encouraged to consult the book by Brownston et al. (1985) for more details.

6.6.3 Inference Engine

The inference engine goes through the following cycle:

1. Match For each rule, find all matching instances. The output of the MATCH routine is known as the *Conflict Set* (CS). If the CS is empty then the program exits. The *Conflict Set* is a made up of elements of the form (*rulename timestamp₁ timestamp₂... timestamp_k*). Where the *timestamp_i* is the sequence number used to identify the *WME* matching the corresponding pattern in the rule. That is, the *Conflict Set* contains a set of rules that are ready to fire along with their matching *WMEs*.

2. Resolve Select one element from the *Conflict Set*. That is, decide what rule is going to fire with its data.

3. Execute Execute (or fire) the selected rule. Make the appropriate changes in the WM, and also do any Input/Out that is indicated. Then go back to step 1.

The most straightforward task in the above cycle is step 3. It simply involves executing the instructions in the RHS of the rule. The other two steps need considerable attention because they determine the problem solving efficiency.

The second step RESOLVE is the one that deploys the problem solving strategy in OPS5. The Match-Resolve-Execute cycle of rule based systems can be viewed as doing hill climbing search in a (dynamic) state space. The *Match* step is equivalent to applying the *moveGen* function (see Chapter 2) to a given state. It generates the possible changes one can make to the state. The *Resolve* step embodies the strategy of search. It decides which of the candidate moves should be applied. That is, which rule in the CS should be selected to fire.

One strategy to select the *move* (rule instance with matching data) is some fixed strategy like the first (in order of writing) rule matching. This would be akin to doing depth first search in the state space. In practice,

rule based systems employ some more sophisticated strategies. For example, *OPS5* uses the following criteria, in order of application, to order the *Conflict Set* (CS).

Refractoriness⁷ This says that a given rule will fire only once with a given combination of data. This prevents the same rule from firing again and again with the same data, when the RHS does not change the data. For example, the rule named *top-card* above will fire only once with the given data.

Recency Choose the rule that matches the most recently produced data. This helps keep a reasoning thread going. The most recently created data (*WME*) gets acted upon quickly.

Specificity This says that given an option between two rules, the rule that has a more specific LHS is preferred. Specificity can be measured by counting the number of attributes in the patterns in the LHS.

The last strategy is known as the LEX strategy. A variation of LEX known as MEA (means ends analysis—see Chapter 7). In MEA, the recency of the first pattern in all the rules is used to select the rule to fire. If there is a tie between rules then the system falls back upon the LEX strategy.

One can use the specificity rule to implement *default rules* which are fired when a more specific rule does not match. If a more specific rule were to match then that would be preferred. For example, in a contract bridge bidding program, one may have a rule asking a player to pass if there is no reason (matching rule) to make any other bid. One could write two rules as follows:

```
{p pass-rule
  {turn-to-bid ^player <x>}
→
  {make (bid ^player <x> ^bidName pass)}
{p 1N-opening
  {turn-to-bid ^player <x>}
  {hand ^player <x> ^points <<15 16 17>> ^shape balanced}
  - (bid)
→
  {make (bid ^player <x> ^round 1 ^type opening
        ^bidName notrump ^denomination 1)}}
```

Given these two rules, if the program encounters a balanced hand with 15 to 17 points⁸ and no one else has bid then it would make an opening bid of *1 no trump* on its turn. Otherwise, the rule named *pass-rule* will advocate discretion and pass. In practice, a *bidding system* would have hundreds of rules.

Apart from the built-in conflict resolution strategies in the language, *OPS5* also allows the user to define priority of rules. One may define different levels of priority classes as follows.

(priority priority-class value).

For example, one could have the following priority classes,

(priority top-level-rule 120)
(priority higher 60)
(priority high 50)
(priority low -50)
(priority lower -60)
(priority lowest -100)

Then one could assign individual rules to any group, as given below:

(rule-priority rule-name priority-class)
(rule-priority 1N-opening high)
(rule-priority Minor-suit-opening low)

The effect of this is that if both rules, *1N-opening* and *Minor-suit-opening*, are in the *Conflict Set* then the *1N-opening* rule will be preferred over the *Minor-suit-opening* rule, irrespective of how the in-built strategies evaluate them.

6.6.4 Some Sample Rules

We look at a few sample rules from (Khemani, 1989). The rules are for a knowledge based program for declarer play in bridge. The rules give some idea of the language, and also illustrate that rules in a program may embody different functionalities. They may represent some knowledge about expert plays and their applicability, they may be used for describing and recognizing patterns, or they may even be used for 'run of the mill' computations and updating of data.

The rule below is used to make an inference about the opponents' cards in a given suit when an opening lead is recognized to be the fourth best card of the player's suit⁹. It also makes an inference about how many cards above the led card the partner of the player has.

```

(p could-be-fourth-best
  (<oc> {outstanding-cards <n>})
  (card-rank ^card <c> ^played lead ^player <west> ^rank <r> ^suit <suit>)
  (partner ^of <west> ^is <east>)
  (card-rank ^player << north south >> ^suit <suit> ^played no
    ^rank {<rc> < <r>} ; number of cards higher
    ^int-rank {<ir> <= <n>}} ; than <c> is <= <n>
  -(card-rank ^player << north south >> ^suit <suit>
    ^rank {< <r> > <rc>} ^played no)
  (opponents ^hold {<kk> >= 3} ^cards-of <suit>) ; 1 lead
-->
  (remove <oc>)
  (bind <diff> (compute <n> - <ir>))
  (write record ..could be fourth best..)
  (make inference ^player <east> ^has-cards <diff>
    ^above <c> ^rank <r> ^suit <suit>)
  (make inference ^player <west> ^has-cards 3
    ^above <c> ^rank <r> ^suit <suit>)
  (make inference ^suit <suit>
    ^breaks (compute 1 + <kk> - <diff>)
    ^and <diff> ^or better)
  (make type-of ^lead fourth-best))
(rule-priority could-be-fourth-best opponents-card)

```

The second rule is used to recognize an honour card lead, and also makes inferences about holdings in that suit.

```

(p honour-led
  (card-rank ^card << a k q j >> ^played lead ^player <west>
    ^suit <suit> ^rank <rank>)
  (one ^plus <rank> ^is <second>)
  (card-rank ^player ew ^rank <second> ^suit <suit> ^card <jack>)
  (one ^plus <second> ^is <third>)
  (card-rank ^player ew ^rank <third> ^suit <suit> ^card <ten>)
-->
  (make type-of ^lead honour)
  (write record ..an honour lead..
    <west> should have the <jack> and <ten> as well..)
  (make inference ^player <west> ^has <ten>
    ^suit <suit> ^rank <third>) ;weaker inference ??
  (make inference ^player <west> ^has <jack>
    ^suit <suit> ^rank <second>))
(rule-priority honour-led opponents-card)

```

Before one plans the play, one has to do a survey of one's resources to decide which strategies are going to be feasible. The following rule establishes some of the links between the two hands of the partnership.

```

(p constructing-entry-map
  (pick ^patterns <suit>)
  (card-rank ^player {<player> << north south >>}}
    ^rankclass 1 ^rank <rank>
    ^card <card> ^suit <suit> ^played no)
  (partner ^of <player> ^is <partner>)
  (card-rank ^player <partner> ^card <entry>
    ^played no ^suit <suit> ^rank {<r> > <rank>}})
  -(entry ^from <entry> ^to <card> ^of <suit>)
  -->
  (make entry ^from <entry> ^to <card> ^of <suit>
    ^hand <player> ^rank <r> ^used no ^played no))
(rule-priority constructing-entry-map pick-patterns)

```

The next rule is a rule that suggests a strategy in play, known as a *Bath coup*. The strategy is applicable only in certain situations.

```

(p try-bath-coup
  (card-rank ^played lead ^player <west> ^suit <suit> ^rank 2)
  -(lead-trick ^of <suit>})
  (solidity ^suit <suit> ^rankclass 1 ^cards 1)
  (solidity ^suit <suit> ^rankclass 4 ^cards 1)
  (sequence-is ^plays <west> ^after <south>})
  (card-rank ^suit <suit> ^player <south> ^rank 4 ^played no)
  (holding ^suit <suit> ^trc 1 ^length > 2)
  (holding ^player <south> ^suit <suit> ^length > 2)
  (tricks ^leeway >= 1)
  -->
  (write record ..let me try a bath coup..)
  (make encourage <suit>})
  (make lead-trick ^of <suit>})
  (make duck ^rounds 1 ^of <suit>))
(rule-priority try-bath-coup opponents-card)

```

The following rule takes over after all players have played a card, and decides the winner of the trick.

```

(p deciding-winner
  (context ^for playing)
  (four cards over)
  (<f> {follow-suit ^to {<suit> << spades hearts diamonds clubs >>}}})
  (current-trick ^suit {<suit> << spades hearts diamonds clubs >>}}
    ^card <card> ^rank <rank> ^player <winner>})
  -(current-trick ^suit <suit> ^rank < <rank>})
  (trick ^number <n>})

```

```

-->
  {make winner ^of-trick <n> ^player <winner>
    ^card <card> ^suit <suit>}
  {write record (crlf) (tabto 20)
    TRICK <n> WAS WON BY <winner> (crlf)
  {tabto 20) WITH THE <card> OF <suit> (crlf))
  {remove <f>}}
{rule-priority deciding-winner trick-accounting)

```

6.6.5 Matching Rules with Data

Let us now turn our attention to the most demanding task that the inference engine has to do. This constitutes constructing the *Conflict Set* from which the rule to be executed will be chosen. For the sake of completeness, it must pick all the matching rules with all matching permutations of data. Consider the rule with the following LHS:

```

{p some-rule
  {card-rank ^rank > 1}
→
  {some actions}}

```

Then given a pack of 52 cards, it will match all cards that are not aces, that is, 48 of the 52 cards. Thus, 48 instances of this rule will find their way into the *Conflict Set*. Also, the matching algorithm will have tried this rule on all 52 cards and selected 48 out of them. If our algorithm is a simple brute force algorithm then in fact, the rule would have tried matching all the data in the working memory before selecting the 48 instances.

In the general case, if a rule has K patterns then each of the patterns will have to be tried with each *WME*. Thus, if there are R rules, each with K and the working memory has M *WMEs*, the brute force match algorithm will have to do $M \cdot R \cdot K$ pattern comparisons.

And it will have to do that in each Match-Resolve-Execute cycle because the actions of the selected rule make changes in the working memory.

It has been empirically observed that the *Match* component uses up to eighty to ninety percent of computation time in rule based production systems. Therefore, it would make sense to improve upon the efficiency of the *Match* algorithm. That is precisely what was done by Charles Forgy when he implemented the OPS5 language. He designed a network, called the *Rete* network that vastly improved the performance of the *Match algorithm*.

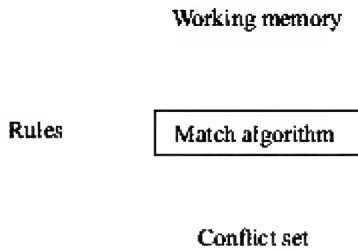


Figure 6.18 The Match algorithm takes the set of rules and the set of working memory elements and generates the *Conflict Set*. The rules and the working memory are independent representations.

6.6.6 Rete Algorithm

Consider the brute force algorithm mentioned above. Schematically, it is a procedure that takes the set of rules and the working memory as an input, and generates the *Conflict Set*, as depicted in Figure 6.18.

The rules and the working memory are stored separately. The match algorithm tries to match all rules with all permutations of *WMEs*. Matching each rule involves matching a pattern with a *WME*, which in turn involves testing for matching attributes within each pattern.

The *Rete*¹⁰ match algorithm (Forgy, 1982) is based on the following observations.

1. Many patterns share the tests to be done. The same pattern may occur in different rules. It will be tested separately for each rule by the brute force algorithm. If possible, this test should be made only once and the result should be available to the different rules. Secondly, patterns that are not identical may still share some tests. If possible, these tests should also be done commonly as few times as possible. For example, the two patterns,

(card-rank ^suit spades ^player south ^rank 1), and
 (card-rank ^suit spades ^player south ^rank 3)

share the first two attributes and values. Then, for any given *WME*, these two tests could just be done once.

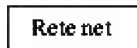
2. When a rule fires, it makes only a few changes in the working memory. It may remove a few *WMEs* and add¹¹ a few *WMEs*. This means that many rules that were in the *Conflict Set* would continue to do so, with the same set of matching *WMEs*. The brute force algorithm will compute these matches again in the next cycle. This can be avoided, if we carry forward the previous CS and only make changes to it.

The *Rete network* is a compilation of the rules that also serves as distributed memory for *WMEs*. Each *WME* resides at some node in the network depending upon the matches that it has made and the constraints with other patterns that it has satisfied. Since a *WME*, or part of *WME*, can match multiple rules, it is possible that several copies of a *WME* will be

made and will reside at different nodes. The *Rete* algorithm may, therefore, have a larger space requirement than the brute force matching algorithm. But the reduction in time complexity more than makes up for the increased space requirements.

The *Rete* algorithm uses the *Rete* network. It accepts changes in the Working Memory, and generates changes in the *Conflict Set*. The working memory changes are input in the form of tokens of the form $\langle +WME \rangle$ and $\langle -WME \rangle$. The positive sign means that a new *WME* has been created, and a negative one means that a *WME* has been removed. Remember that these are the two kinds of changes the RHS of a rule can make. The tokens traverse the net and eventually may combine with existing tokens to satisfy the LHS of a rule. The rule will then get added to the *Conflict Set*. Negative tokens may go and cancel some existing token, and result in the removal of a rule from the *Conflict Set*. The effect of the tokens is reversed for patterns with the negative sign. For such patterns, a positive token may remove a rule from the CS, while a negative token may insert a rule into it. The function implemented by the *Rete* algorithm is depicted in Figure 6.19.

Changes in WM



Changes in CS

Figure 6.19 The *Rete* match algorithm compiles the rules into a network. It accepts changes in the Working Memory as input, and computes the resulting changes in an existing possibly empty *Conflict Set*.

The positive or negative tokens are input at the top of the net and the traverse downwards towards the rules. The net is made up of two kinds of nodes.

The top half of the network is a discrimination net that switches a token towards the rules that it matches. It is made up of nodes call *alpha* nodes. Each *alpha* node tests the value of one attribute and accepts tokens that satisfy its conditions. The token in question resides at the predecessor node before the test. The predecessor may itself be an *alpha* node or it may be the root node. Different *alpha* successors of a given node, test for different conditions. Figure 6.20 below illustrates the notion of an *alpha* node. In this figure, there are three *alpha* nodes and two example tokens.

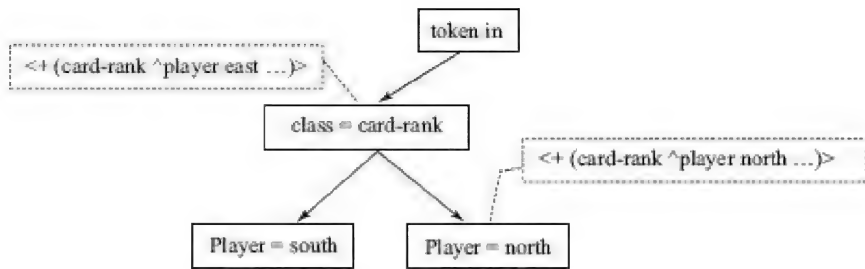


Figure 6.20 Alpha nodes, shown in the tree structure, test for conditions of attributes. In the figure, the two tokens, shown in dashed boxes, will reside at the two nodes as shown.

The first token `<+(card-rank ^player east ...)>` satisfies the attribute test for `class = card-rank`, but does not match the values of the attribute “player” for any of its children. It will thus stay here, and never match any rule in this network. The second token matches two tests in the discrimination tree. If some other attribute’s value in this token matches some child, it will travel further down the tree.

After the tokens have traversed across all *alpha* nodes, they encounter *beta* nodes in the lower part of the network.

The lower part of the network is an assimilative network made up of nodes called *beta* nodes. Each *beta* node, shown as circles in Figure 6.21 below, pulls tokens from its two ancestors without performing any check. If there is more than one *beta* child of a node, each gets a copy of the token. However, it lets the token go forward only if it can pair up with a matching token from the other branch. The two tokens match if any variables shared by the two patterns are bound to the same values. For example, if the two tokens that arrive at node B_1 (as shown) are,

`<+(card-rank ^player south)>`, and
`<+(turn ^of south)>`

then the *beta* node B_1 packs the two into a larger token and allows it to pass, when the node B_4 pulls the compound token, and waits for another token from the left branch. Meanwhile, even node B_3 pulls a copy of the token `<+(card-rank ^player south)>` from the parent *alpha* node, but it has to wait for a token to arrive from its other parent B_2 . When a matching token does arrive then B_3 would have assembled the complete matching LHS for rule *RuleName*. The rule can then be added to the *Conflict Set*.

Each *alpha* node has one parent, but may have many children. The parent has to be an *alpha* node or the root. The children of *alpha* nodes can be either *alpha* or *beta* nodes. Beta nodes, on the other hand, have two parents¹². They could be either *alpha* nodes or *beta* nodes. Beta nodes have one or more children, which are either *beta* nodes or rule nodes. When a *beta* node has collected all the tokens for satisfying the LHS of the rule below, it passes the tokens to the rule node, which creates an instance of the rule to add to the *Conflict Set*.

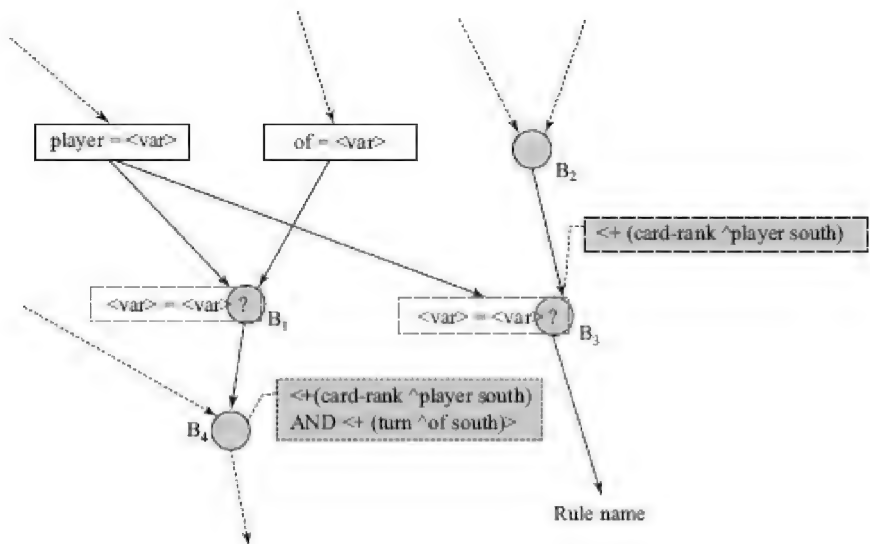


Figure 6.21 Beta nodes or join nodes combine two tokens when the variable condition (if any) matches and allow them to pass down. In the figure, there are two tokens $\langle +(\text{card-rank } \wedge \text{player south}) \rangle$ and $\langle +(\text{turn } \wedge \text{of south}) \rangle$. Note that there are two copies of the first token, one at B₃ and one at B₄.

The top half of the *Rete* net sorts or *classifies* the tokens according to the test patterns in the rules. It acts like a switch that routes tokens towards their target rules. The bottom half of the network assimilates the tokens required by a rule, and hands them over to the rule node.

6.6.7 Execution

First the rules are compiled into the network. Then, the following cycle ensues.

1. Insert a bunch of tokens into the root (token in) node of the net. If there are no tokens, then exit the *Match* phase. Each token traverses down the matching *alpha* nodes and waits for its partner at a *beta* node. If multiple *beta* nodes want the token then copies are made. When the partner of the token arrives, they form a compound token and travel down together. The last stage is the rule node where the matching tokens of a rule arrive.
2. The conflict resolution strategy is applied and a rule is selected to fire. If the *Conflict Set* is empty then execution ends.
3. The selected rule is executed. A bunch of new positive and negative tokens are created. Control returns to step 1.

The above description does not contain the details about how negative tokens are handled. This is left as an exercise for the reader. Also, it seems that the author of the algorithm has subsequently developed improved versions said to be several orders of magnitude faster, but which are not available in the literature¹³. The reader would find it interesting to

work upon improvements to the algorithm. The data-driven processing on a network of nodes makes it a promising candidate for implementing OPS5 on a parallel machine, and one can find more detail in (Gupta et al., 1986; 1989).

We next look at a toy example and the corresponding *Rete* net.

6.6.8 An Example

Let us say that we have data on objects from the blocks world, described in terms of the shape, size and colour of the base, sides and top. The following rules are used to recognize one of pyramids, cylinders, wands and domes. For example, the wand rule¹⁴ (number 3) says that,

```
IF          The block has name <x>
    &       The base of the block is circular with area 1
    &       The side of the block is curved and tapering with colour
            black
    &       The top of the block is pointed
THEN
```

Classify block named <x> as a wand.

Let us say that we have these four rules.

```
Rule 1: (p greenPyramid-rule
        {block ^name <x>}
        {base ^block <x> ^shape square ^area > 1}
        {side ^block <x> ^angle < 90 ^surface plane ^colour green}
        {top ^block <x> ^surface point})
→
  {make {class ^block <x> ^type greenPyramid}}}
```

```
Rule 2: (p cylinder-rule
        {block ^name <x>}
        {base ^block <x> ^shape circle ^area > 1}
        {side ^block <x> ^angle 90 ^surface curved}
        {top ^block <x> ^surface flat})
→
  {make {class ^block <x> ^type cylinder}}}
```

```
Rule 3: (p wand-rule
        {block ^name <x>}
        {base ^block <x> ^shape circle ^area 1}
        {side ^block <x> ^angle < 90 ^surface curved ^colour black}
        {top ^block <x> ^surface point})
→
  {make {class ^block <x> ^type wand}}}
```

```

Rule 4: (p dome-rule
        (block ^name <x>)
        (base ^block <x> ^shape circle ^area > 1)
        (side ^block <x> ^angle 90 ^surface curved)
        (top ^block <x> ^surface spherical)
→
  (make (class ^block <x> ^type dome)))

```

We construct the *Rete* net for the rules above. Observe that there is no unique *Rete* net corresponding to a set of rules. This is because one could choose any order of tests while constructing *alpha* nodes and any order of combining in *beta* networks. The objective will be to construct a network where maximum attribute value tests are shared. The following example net (Figure 6.22) is one possible network. The network also stores the *WMEs* described below.

1. (base ^block A ^shape square ^area 20)
2. (base ^block B ^shape circle ^area 20)
3. (base ^block C ^shape circle ^area 1)
4. (side ^block C ^angle 85 ^surface curved ^colour black)
5. (side ^block A ^angle 45 ^surface plane ^colour green)
6. (top ^block A ^surface point)
7. (top ^block C ^surface point)
8. (top ^block D ^surface point)
9. (side ^block B ^angle 90 ^surface curved)
10. (top ^block B ^surface flat)
11. (block ^name A)
12. (block ^name B)
13. (block ^name D)

The numbers on the left are recency numbers, stating the order in which these *WMEs* are created. They are used to identify the *WMEs* in the *Rete* net. Observe that some tokens have been combined in *beta* nodes. The variable and the value for each such join event is also shown. Finally, one can see that in his example, there are two rules in the *Conflict Set*.

6.7 Discussion

Problem decomposition involves breaking up a problem into smaller parts and working them. In backward or goal-directed reasoning, we naturally break up the goal into subgoals and tackle each separately. This leads to the And/Or search for finding the decompositions that make up the solution. We will revisit goal-directed reasoning again in Chapter 7 on planning, and Chapter 12 on logic and theorem proving.

The mechanism used for problem decomposition was production systems, in which each goal is replaced by a set of subgoals and the subgoals have choices for solving them. But the process of reasoning in production systems is aligned with the arrow, from the left-hand side to the right-hand side. That is, the forward direction. We could use the *forward*

The lure of rule based systems is that the task of building problem solvers can be divided between different people. The computing community can develop the inference engine, and the domain personnel can write the “business logic” in the form of rules. This is slightly different from the situation depicted in Figure 2.2, where the “user” had to implement the domain functions. Here, the user is encoding the problem solving strategy used for solving problems in the domain.

The question of forward reasoning versus backward reasoning still remains. At an abstract level, one can reason that the choice could be resolved based on the nature of the search graph. If there is a lot of data narrowing down to a few goals, it might be appropriate to use forward reasoning. If, on the other hand, the number of possible goals is very high, it might be more appropriate to start reasoning with the goals that one has to solve. It has been observed by many including Robert Kowalski who formalized the notion of knowledge based systems (Kowalski, 1979), that human beings very often do backward reasoning.

Some clues on this matter come from the domain of visual perception. Visual perception is the task of making sense of the excitatory signals that a living creature receives on its retina in the eye. To quote the neuroscientist V.S. Ramachandran (2003)—“*Our ability to perceive the world around us seems so effortless that we tend to take it for granted. But just think of what is involved. You have two tiny, upside-down distorted images inside your eyeballs, but what you see is a vivid three dimensional world out there in front of you*”. About one third of our brain comprising thirty different regions in the visual cortex is devoted to visual processing. There are different areas for recognition of colour, different areas for motion, and different ones for recognizing faces.

Scientists have been able to identify these areas because once in a while a patient suffers damage to a part of the cortex and then they are unable to do that kind of processing. So while one patient cannot even recognize his mother’s face even when the voice is recognized, another one is afraid to cross the road because she cannot judge the speed of a car but can very well read the number plate on it. Furthermore, it seems there are two pathways from the retina into the brain for these signals to travel. One, known as the “old” pathway, is the evolutionary ancient pathway that seems to function more in what we would call the ‘forward direction’—collecting and assimilating signals and acting upon them. The second, or the “new”, goes to the visual cortex at the back of the brain. The cortex is the area of our brains that does symbolic processing, where goals and concepts and all kinds of knowledge resides. This region seems to be processing visual information in the backward direction; that is hypothesising goals¹⁵ of what we are seeing and looking for confirmation of these goals in the incoming data.

Optical illusions are a stark example of the fact that we do backward reasoning in visual processing. For example, in the well known optical illusion in which we can see a young woman or an old woman depicted in Figure 6.23. This image is from an anonymous German postcard from

1888 and depicts the image in its earliest known form.¹⁶



Figure 6.23 Optical illusions illustrate the fact that what we see is what we believe we are seeing. This famous optical image in which you can switch between seeing a young woman or an old woman, is one of the oldest known.

The interesting thing is that we seem to process our visual imagery in both forms of reasoning, forward and backward. This was revealed by an experiment in which a patient had his second pathway to the right visual cortex damaged. The patient was asked to point his finger to a point he said he could not see, but when asked to try, he could do it ninety nine percent of the time. So our ancient pathway leads to an area of the brain where processing happens in the forward direction. This kind of processing does happen in simpler life forms more often. For example, the visual system of a frog is tuned to recognize flying insects. It can only see an insect when it moves, and is blind to a stationary one.

However, recent research shows that with the development of our neocortex, or the cerebral cortex, and our ability to handle a large number of concepts, the number of objects and situations we might be seeing is so large that we seem to be doing perception more in a goal directed fashion. We seem to hypothesise what we are seeing and only use the incoming visual information to validate our hypothesis. The following interesting experiment was discussed in a BBC program on vision¹⁷—“Almost a third of our entire brain is devoted to vision, but there’s a limit to how much our eyes observe. In a simple experiment in a busy shopping centre, Nigel asks shoppers for directions and then switches places with someone else while the shopper is distracted. Most people failed to notice the switch and carried on giving directions. Dr Richard Wiseman of the University of Hertfordshire explains that people’s brains weren’t paying attention to who was asking the questions because they were just concentrating on getting the directions right. So, ultimately, we only see what our brains want us to see.”

This chapter reinforces our hypothesis that knowledge is a key to

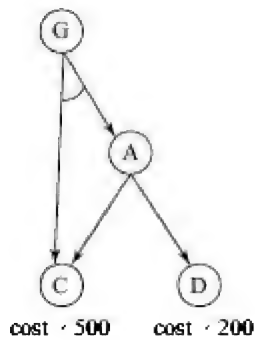


Figure 6.25 A problem with nonzero cost of *SOLVED* nodes.

5. In the AO graph of Figure 6.26, the *SOLVED* nodes along with their costs are shown with double circles. Labels on internal nodes are heuristic values. Assume each edge has a cost 10. Show how the algorithm AO* will solve this graph. Show each expansion clearly and highlight the final solution. Is the algorithm guaranteed to find the optimal solution for this problem? Justify your answer.

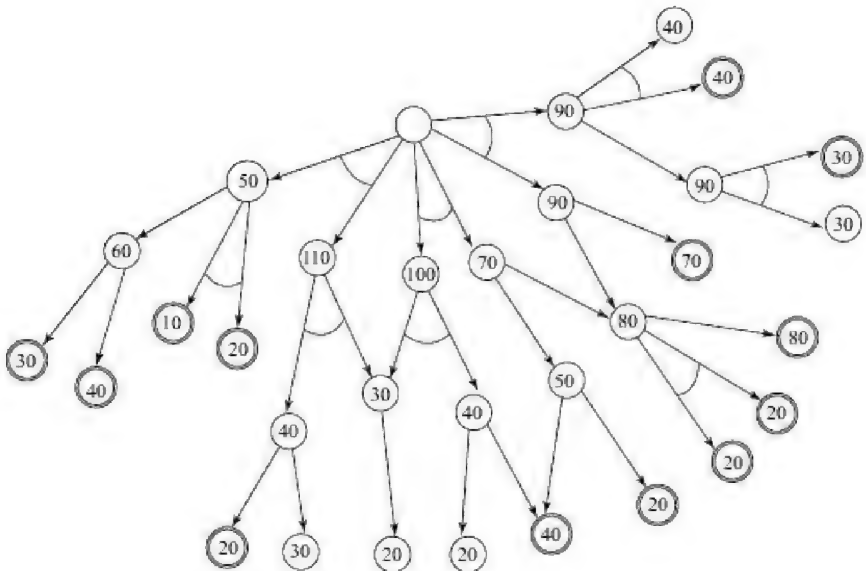


Figure 6.26 Another AO graph. The solved nodes, shown in double circles, have a cost associated with them. Values in internal nodes are heuristic estimates of cost.

6. The problem graph in Figure 6.27 is similar to the graph in Figure 6.26, but with a different set of values. How would AO* perform on the new problem?

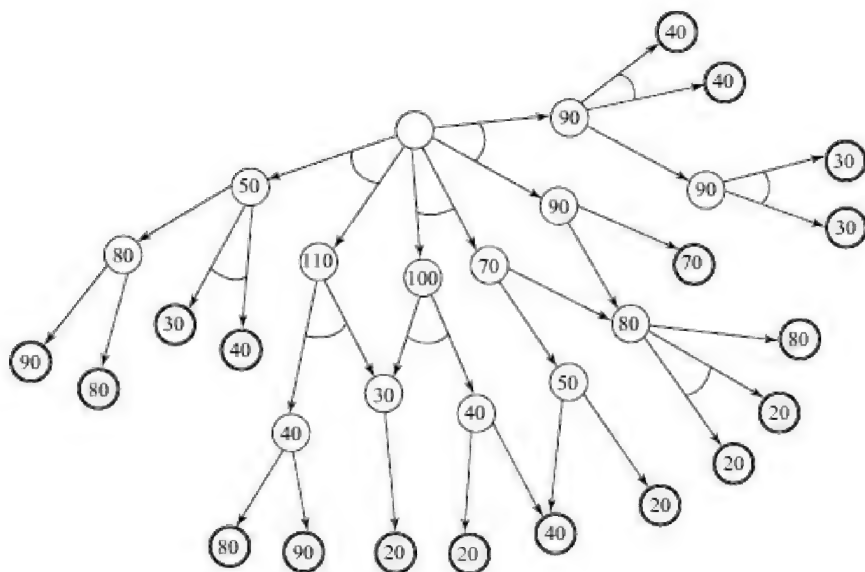


Figure 6.27 The problem from Figure 6.26 with a different set of values.

7. *Rete Net*. Write a program to read in a set of rules in a given format and construct a *Rete Net* for the rules. Show the *Rete Net* in a graphical form. Note that each node must have an associated memory to store tokens. The user should be able to query for the instances of matching rules.
8. Forward Chaining System. Read a set of rules (program) from a file (or the keyboard). Call the program for constructing the *Rete Net*. Read a set of working memory elements from a file (or the keyboard). Accept a command (*run N*), and execute *N* cycles of Match-Resolve-Execute. Accept a command (*run*) to do the same in an infinite loop. Include the following set of actions in rules. (1) Read—a file containing *WMEs*, (2) Print—a message on the screen (or a file), and (3) Halt—stop the cycle. In a debug mode, it should show the *Conflict Set* after each step. Implement two conflict-resolution strategies.
9. Write the most appropriate right-hand side for the following rule:


```

      {p what-does-this-rule-do
        (inst ^number <n>)
        -(inst ^number <m> << <n>)}
      →
        {assert . . .}}
      
```
10. Given a set of three *WMEs* of the form,

(Marks subject: physics student: satish scored: 96) separately for physics, chemistry and math, and one *WME* for each student initialized to values as shown below,

(Total name: satish rank: 0 totalMarks: -1)

Write a set of production rules to sum up the marks for each student and store it in the attribute *totalMarks:* of the *Total WME*, and assign a

rank to each student in the attribute *rank*: based on marks. The number of students is not known and students with the same total marks may be ranked arbitrarily.

11. Write the conditions for the rules in English, and construct a Rete net for the following set of rules:

```

Rule 1: (p fare-infant
        (inst ^person <x> ^type child)
        (age ^of <x> ^is < 5)
        (inst ^person <y> <> <x>) {y not equal to x}
        (age ^of <y> ^is >= 5)
        (traveling ^person <x> ^with <y> ^to <xyz>)
→
    (some Right-Hand Side))

Rule 2: (p fare-child
        (inst ^person <x> ^type child)
        (age ^of <x> ^is < 12)
        (age ^of <x> ^is >= 5)
        (traveling ^person <x> ^to <anywhere>)
→
    (some Right-Hand Side))

Rule 3: (p fare-student
        (inst ^person <x> ^type student)
        (age ^of <x> ^is < 25)
        (traveling ^person <x> ^to <<home school>>)) (or)
→
    (some Right-Hand Side))

Rule 4: (p fare-games
        (inst ^person <x> ^type sportsman)
        (traveling ^person <x> ^to games)
→
    (some Right-Hand Side))

Rule 5: (p fare-arjuna
        (inst ^person <x> ^type sportsman)
        (traveling ^person <x> ^to <anywhere>)
        (awardee ^person <x> ^award Arjuna)
→
    (some Right-Hand Side))

```

12. Construct a *Rete* net for the following set of rules:

```

        (age of <y> is {<ageY> > <ageX>}) /* <y> is older than <x> */
→
    (make (tau ^of <z> ^is <y>))
Rule 3: (p younger-Than
        (age of <x> is {<ageX>})
        (age of <y> is {<ageY> > <ageX>})
→
    (make (younger ^than <y> ^is <x>))

```

In the *Rete* net show, where the tokens for the following data will reside, and what is the *Conflict Set*?

- Token 1: (age ^of Ramu ^is 17)
- Token 2: (age ^of Ramesh is 36)
- Token 3: (brother ^of Mahesh ^is Ramesh)
- Token 4: (son ^of Suresh ^is Ramesh)
- Token 5: (son ^of Mahesh is Ramu)

13. Write a rule to define transitivity of the “younger” relation.
14. A typical bank manager may decide to sanction a loan to an applicant based on some criteria such as creditworthiness, source of income, existing loans, nature of job, salary, etc. Define a set of attributes that a bank manager might use and write a few sample rules.
15. Write a rule based program to make a move in the Cross&Noughts (or tic-tac-toe) game.
16. Consider the task of writing a piece of software (or an “app”) for assisting decision making in a medical emergency. For example, one might be able to prescribe some action immediately (loosen the clothes; or sprinkle water). Or one may be able to say that the person needs to be taken to a hospital immediately. Design a few sample rules that would go into such a system.

-
- ¹ To look at an extreme example, graphite and diamonds are just different arrangements (allotropes) of carbon atoms.
 - ² The Joshua Lederberg Papers, Computers, Artificial Intelligence, and Expert Systems in Biomedical Research, in Profiles In Science, National Library of Medicine, available at <http://profiles.nlm.nih.gov/BB/Views/Exhibit/narrative/ai.html>
 - ³ The so called “Official Production System” language was developed by Charles Forgy in Carnegie Mellon University.
 - ⁴ McDermott’s 1980 paper on R1 won the AAAI Classic Paper Award in 1999. According to legend, the name of R1 comes from McDermott, who supposedly said as he was writing it, “*Three years ago I wanted to be a knowledge engineer, and today I am one.*” —<http://en.wikipedia.org/wiki/Xcon>
 - ⁵ “VAX” was originally an acronym for *Virtual Address eXtension*, both because the VAX was seen as a 32-bit extension of the older 16-bit PDP-11 and because it was a commercial pioneer in using virtual memory to manage this larger address space.—<http://en.wikipedia.org/wiki/VAX>
 - ⁶ Traditionally, in contract bridge literature, the four players are called North, South, East and West.
 - ⁷ This term comes from the neurobiological observation of a refractory period for a neuron, which means that the neuron is not able to fire immediately without first going through a relaxation process. — OPS5 Reference, available at <http://www.cs.gordon.edu/local/courses/cs323/OPS5/ops5.html>
 - ⁸ The author strongly advises the reader to learn the game of contract bridge.
 - ⁹ Players often choose the cards to play in accordance to stated conventions that are designed to convey information to the partner. Since the conventions are known, this also means that opponents are privy to this information too.
 - ¹⁰ The Latin word for net is *rete*.

- 11 Modifying a *WME* can be seen as removing it and adding a new one.
- 12 We can easily generalize this to have more parents.
- 13 Apparently for commercial reasons. —
http://en.wikipedia.org/wiki/Rete_algorithm
- 14 From a secret rule book at Hogwarts not accessible to muggles.
- 15 And therefore, when people say to one another “you see only what you want to see” they are probably right.
- 16 *<http://mathworld.wolfram.com/YoungGirl-OldWomanIllusion.html>*
- 17 *<http://www.bbc.co.uk/science/humanbody/tv/humansenses/programme2.shtml>*

Planning

Chapter 7

An agent interacts with the world via perception and actions (Figure 7.1). Perception involves sensing the world and assessing the situation. In sophisticated systems, it may involve creating some internal representation of the world, which we can call a mental model. Actions are what the agent does in the domain. Planning involves reasoning about actions that the agent intends to carry out. “*Planning is the reasoning side of acting*” (Ghallab et al., 2004). This reasoning involves the representation of the world that the agent has, as also the representation of its own actions.

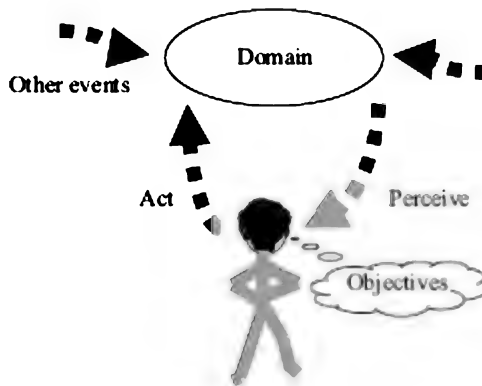


Figure 7.1 A planning agent can perceive the world, and produces actions designed to achieve its objectives. In a static domain, the agent is the only one who acts. A dynamic domain can be modelled by other agencies that can change the world.

The objectives of planning are to achieve some desired situation. The objectives could be certain properties of the final state after all actions have been completed, or even along the way in the sequence of states. These could be strict requirements or hard constraints where the objectives *have to* be achieved completely for success. The objectives could also be soft constraints, or *preferences*, to be achieved to the extent possible.

The planning problem can be described or posed at varying levels of detail. As the detail increases, so does the complexity of the computation. In the simplest planning problems, the domain is static, the agent has complete information of the domain (perception is perfect), actions are instantaneous and their effects are deterministic. The agent knows the world completely, and it can take all facts into account while planning. It is the only one changing the world, and therefore does not have to worry about the world changing while it is planning or executing a plan. The fact that actions are

instantaneous implies that there is no notion of time, but only of sequencing of actions. The effects of actions are deterministic, and therefore the agent knows what the world will be like after each action. Each of these constraints can be relaxed to define richer planning domains, but in which planning is a computationally harder task.

The algorithms being developed in the planning community are all based on explicit representations of actions in a domain. This is a little different from our earlier view of search which was more abstract and tended to assume a move generating function. Representing actions explicitly allows us to explore richer domains, for example when actions have durations. It also allows us to investigate approaches to planning different from graph search.

In *classical planning*, reasoning about actions involves simulation of changes in the mental model that the agent has. The agent peers into the future to examine the different possibilities and select the desirable course of action. This is called *projection* into the future. The planner searches through the possible combination of actions to find the *plan* that will work.

An alternative approach is to exploit the agent's experience, looking into the past. If the agent has memory, it can reuse a plan that worked earlier, or reuse the problem solving experience to find a solution. The agent can fish out (retrieve) a plan or planning cues from its memory. Such an approach is known as *memory based planning*. It involves representation and reasoning over the events in the past. Human beings typically use both methods. If a problem is familiar then a known solution is used from memory. The idea is to not reinvent the wheel every time. However, when a problem is a new one, the first principles projection based methods have to be used. An efficient agent, for example an intelligent human being, will keep learning from experience, as and when new problems are encountered and solved. We will explore memory and knowledge based techniques later in the book (see Chapter 15). For the moment, we focus on how to find plans from first principles, by projecting the effects of candidate actions into the future.

In order to explore planning algorithms in a domain independent form, a standardized notation for describing the world and also the actions has been developed. This is known as the *Planning Domain Description Language* (PDDL) (McDermott, 1998). PDDL allows one to define the language in terms of which the world will be described, as well as the actions that the planner is capable of, expressed in terms of schemas. The language is based on First Order Logic notation (see Chapter 12). It also allows one to state a specific planning problem by describing the current state and the objectives or goals using the constructs defined in the schema. The planning algorithms can then be written in a domain independent form. The domain and problem descriptions¹ will become inputs to the domain independent algorithms.

The complexity of planning will depend upon the expressiveness of the PDDL and the constructs defined. Starting with the simplest planning domain and problems, a series of more expressive PDDL languages have been defined (Fox and Long, 2003; Edelkamp, 2004; Gerevini and Long, 2005). In each new language, the world and the actions can be described in a richer fashion, taking more and more aspects from the real world problem. However, even the simplest of planning domains—known as *STRIPS*

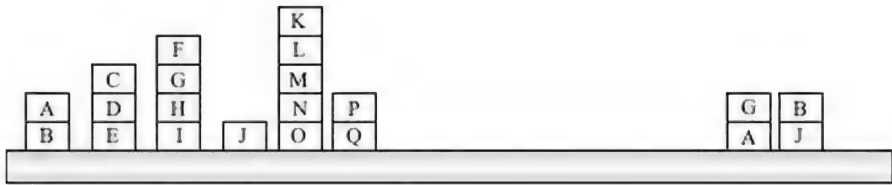
domains described below—pose problems that are computationally hard to solve (Gupta and Nau, 1992). This is not surprising, given that they are basically viewed as search problems in the classical approaches. Nevertheless, there has been considerable interest in planning, and we shall look at some of the vast gamut of techniques that have been developed in the area.

7.1 The *STRIPS* Domain

Planning refers to the choice of moves an agent makes before making the moves in the domain. It involves synthesizing a set of actions, often arranged in a linear sequence, but sometimes also in a partial order. A partial ordering could mean that any linearization is a valid plan. In some domains it could also mean that some actions can be done in parallel. When the action sequence is executed in the domain, it has the effect of transforming the world from a given state into a new state, which if the plan works, is a desired or goal state.

The search algorithms we have seen in the preceding chapters can also be used for planning. The algorithms assume that the state description is available in a compact form, and that generated moves produce new modified states. In practice, the state will be described by its components, and a database describes the world in a given state. Consequently, we also amend our notion of a move. Instead of viewing moves as state transitions, we will view them as *operators* that transform a part of a state (see also Chapter 6). Each operator looks for a pattern in the state and, when applied, makes some changes in the state. In doing so, it does implement the state transition function. But the same operators may be applicable over many states, leading to a more compact representation. The operators are close to the idea of production rules, being applicable when the associated preconditions hold. An *action* (or move in the state transition view) is an instance generated from an operator.

The first program to use this formalism was *STRIPS* (Fikes, 1971)². The program was designed to plan in the domain of blocks world. In this domain, a set of labelled blocks are in some configuration to start with. A block can be on another block, or a block can be on the table, and the table has unlimited capacity. The objective of the planner is to rearrange the blocks into a desired (goal) configuration. The output of the planner is the set of moves that brings about this desired rearrangement. Figure 7.2 illustrates the domain with an example problem.



The given state

The goal = $on(G, A) \wedge on(B, J)$

Figure 7.2 The blocks world domain. Observe that the goal state is incompletely specified. The given state has to be specified completely.

The world, in the blocks world domain, is described by a set of statements that conform to the following predicate schema:

$on(X, Y)$:	block X is on block Y
$ontable(X)$:	block X is on the table
$clear(X)$:	no block is on block X
$holding(X)$:	the robot arm is holding X
$armempty$:	the robot arm is not holding anything

One can make the following observations about the description. There are no metrics involved. A block can have only one block on it, and there is no notion of size, weight or location co-ordinates. We assume an arbitrarily large table with no space constraints. For all purposes, the blocks are all of the same size and shape. They can be handled identically, and only one can rest on another. The one-armed robot can hold one block. The description is essentially qualitative in nature and does not involve numbers in any form.

The representation borrows from logic, specifically first order logic. Unlike in classical logic, however, predicates in planning such as $on(C, D)$ can be true at one time instance, and false at another. Such predicates are also known as *fluents*. It was natural that the first approaches to planning too borrowed from logic, and viewed planning as an exercise in theorem proving (Green, 1970).

Classical reasoning with first order logic does not allow for assertions to be withdrawn. Once an assertion is true, it remains true. There is no notion of time and change. For example, the Pythagoras theorem is true, independently of time. This is fine in the world of mathematics. In a world where an agent is trying to bring about change, one has to have a mechanism for representing situations. For example, in the start state in the illustration above $on(C, D)$ is true. If I pick up (or *unstack*) C and place it on J then $on(C, J)$ will become true. What shall one do with the assertion $on(C, D)$, now that it is no longer true? One way to handle the problem is to introduce time as a parameter, and have every predicate time stamped. Then $on(C, D, t_0)$ would mean that block C is on block A at time t_0 . Then one can have another fact $on(C, J, t_2)$ that states that C is on J at time t_2 . At time t_1 , the arm is holding J , represented by $holding(J, t_1)$. One would also have to

describe goals using variables for time. In Figure 7.2 for example, one would have to describe the goal as “there is some time t at which $on(G, A, t)$ and $on(B, J, t)$ are simultaneously true.”

However, this creates an enormous book-keeping problem of carrying forward facts as time moves on. For example, $ontable(B)$ and all other unchanging facts will need to be asserted afresh as being true at each time step. How do we know that if a fact like $ontable(B, t_1)$ is true then $ontable(B, t_2)$ will be true as well? This is the well known *Frame Problem* introduced by John McCarthy (McCarthy, 1969)³. This problem can be solved by adding the so called “frame axioms”, which explicitly specify that all conditions not affected by actions are not changed while executing that action. Thus, the frame axioms need to be applied at each stage to keep track of what is true as time marches on. We will look at frame axioms in Chapter 10.

The ingenious solution introduced in *STRIPS* was to make *all change explicit*. Thus, one was saved the bother of asserting facts all over again, each time an action was added to the plan. This is the approach a cartoon animator would take making cartoon films. She would take (copy) the previous frame, delete a few strokes, and add a few others to create the next frame. Whatever was not changed remained the same. *STRIPS* operators follow this strategy, and are made up of the following three components:

P: Precondition List The assertions needed to be true for the operator to be applicable.

A: Add List The assertions that became true as a consequence of the operator being applied.

D: Delete List The assertions that are no longer true as a consequence of the operator being applied.

The set of assertions in the database at any time describe the world as it is. When an instance of an operator is applied to a database of assertions, some assertions may be deleted, while others may be added. Fikes introduced the following operators for the blocks world domain.

PICKUP (X)

P: $ontable(X) \wedge clear(X) \wedge armempty$

A: $holding(X)$

D: $ontable(X) \wedge armempty$

PUTDOWN (X)

P: $holding(X)$

A: $ontable(X) \wedge armempty$

D: $holding(X)$

UNSTACK (X, Y)

P: $on(X, Y) \wedge clear(X) \wedge armempty$

A: $holding(X) \wedge clear(Y)$

D: $on(X, Y) \wedge armempty$

STACK (X, Y)

P: $\text{holding}(X) \wedge \text{clear}(Y)$
 A: $\text{on}(X, Y) \wedge \text{clear}(X) \wedge \text{armempty}$
 D: $\text{holding}(X) \wedge \text{clear}(Y)$

Notice that when one picks up a block X , either with *STACK* or *PICKUP*, the operators do not delete the assertion $\text{clear}(X)$. When we put it down, the assertion will already exist in the database. This will not cause a problem with the reasoning, and in fact save some computation, and it is anyway not clear what $\text{clear}(X)$ would mean when it is being held.

An encoding of the blocks world domain in PDDL is given in the accompanying Box 7.1. Observe that the notation used there combines the *add list* and the *delete list* into a common set of *effects*. The facts from the delete list are preceded by a “not”, to depict that the fact is not true in the resulting state. This is consistent with the PDDL standard that evolved later. Also shown is the coding of a small planning problem.

Another important observation is that the goal description may not describe the final state completely. In the above illustration in Figure 7.2, nothing is said about the location of blocks A and J . Likewise, it is not stated what if anything is on B and J . There can be many states in which the given goal conditions will be true. As we will see, this also implies that backward reasoning will have to deal with incomplete state descriptions.

Definition

We say that a state S *satisfies* a goal G iff $G \subset S$.

Box 7.1: The Blocks World Domain in PDDL

The blocks world domain may be described using a typed version of PDDL (PDDL3.0) which is given below. This also introduces the notion of variable types. Also note that add list and the delete list from the *STRIPS* notation is combined into an effects list.

```
(define (domain Blocks)  /* Blocks is a domain dealing with block types */
  (:requirements :typing)
  (:types block)

  (:predicates (on ?x - block ?y - block)  /*The predicates used for
    (onTable ?x - block)                    describing the situation */
    (holding ?x - block)
    (clear ?x - block)
    (armempty))
```

```

/* Given below are the four kinds of actions defined in the Blocks
domain */
(:action pickup
  :parameters (?x - block)
  :precondition (and (onTable ?x) (armempty) (clear ?x))
  :effect (and (not (armempty)) (holding ?x) (not (onTable ?x))))

(:action putdown
  :parameters (?x - block)
  :precondition (and (holding ?x))
  :effect (and (not (holding ?x)) (armempty) (onTable ?x)))

(:action stack
  :parameters (?x - block ?y - block)
  :precondition (and (holding ?x) (clear ?y))
  :effect (and (not (holding ?x)) (armempty) (on ?x ?y) (not
    (clear ?y))))

(:action unstack
  :parameters (?x - block ?y - block)
  :precondition (and (on ?x ?y) (clear ?x) (armempty))
  :effect (and (not (on ?x ?y)) (holding ?x) (clear ?y) (not
    (armempty))))
)

```

We now look at some planning algorithms in the *STRIPS* domain. The search algorithms developed in the preceding chapters can easily be adapted to the task of planning. We do so in the next section. After that, we look at other ways of searching for a plan. We look at the possibility of starting the search from the goal description. This is known as backward reasoning/search. We then describe a way of combining the good features of both, forward and backward search, into an algorithm called Goal Stack Planning. We also look at an approach that seeks to fill in the actions constituting the plan in a nonlinear order, filling in actions as and when we spot their requirement. Such an approach is also described by terms like *Plan Space Planning* and *Partial Order Planning*. Following this, we look at some flavours of hierarchical planning that have been developed. In subsequent chapters, we introduce the concepts of domain independent heuristics and look at their applicability in the planning algorithms discussed here.

7.2 Forward State Space Planning

Given a state, and given a set of operators, one can determine the actions (or moves) that can be applied in the state to generate successor states. This corresponds to the implementation of the *moveGen* function described in Chapter 2. Forward State Space Planning (FSSP) then refers to the search algorithms that start with the given state as the start state, generate the set of successor states, and search through them generating more successors till they find a state that satisfies the goal conditions.

Box 7.2: A Blocks World Problem in PDDL

Figure 7.3 below depicts a planning problem from the blocks world domain. A description of the planning problem in PDDL follows. Observe that the domain field refers to the domain description named Blocks.

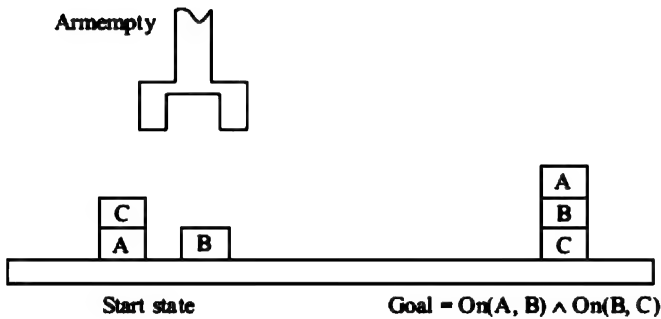


Figure 7.3 A three block planning problem.

The description in PDDL.

```
(define (problem blocksProblem1)
  (:domain Blocks)
  (:objects
    A - block
    B - block
    C - block )
  (:init
    (onTable A)
    (onTable B)
    (on C A)
    (clear B)
    (clear C)
    (armempty))

  (:goal (and (on A B) (on B C)))
)
```

Let $effects(a)$ denote all the effects of action a . Let $effects^+(a)$ denote the set of positive effects of action a . This is the same as the add list in the STRIPS notation. Corresponding to the delete list, we have a similar set denoted by $effects^-(a)$. Then given a state S in which the action a is applicable, the state S' after action a is applied is given by,

$$S' \leftarrow \{S - effects^-(a)\} \cup effects^+(a)$$

That is, from the sets of facts representing S , we delete the facts that are

not true after the action, and add the facts that become true, resulting in the new state S' . We say that the state has *progressed* through the action a . Let us define a function $progress(A, S)$ that returns the successor state when action A is applied to state S .

Given the $progress(A, S)$ function, the search algorithms in Chapter 2 can easily be used. They will, however, suffer from the same drawbacks. The main one being that the search space generated will be huge, and that the search algorithm will have no sense of direction. The given state in Figure 7.2 is described by the following collection of facts:

{ AE , $ontable(B)$, $on(A, B)$, $clear(A)$, $ontable(E)$, $on(D, E)$, $on(C, D)$, $clear(C)$, $ontable(I)$, $on(H, I)$, $on(G, H)$, $on(F, G)$, $clear(F)$, $ontable(J)$, $clear(J)$, $ontable(O)$, $on(N, O)$, $on(M, N)$, $on(L, M)$, $on(K, L)$, $clear(K)$, $ontable(Q)$, $on(P, Q)$, $clear(P)$ }.

In the given state, one instance of the *Pickup* operator is applicable, generating the action $Pickup(J)$. Several instances of the *Unstack* operator are generated. Each of the actions does a small change in the world, and many actions are applicable in turn in the changed world. Figure 7.4 illustrates the fact that the search space generated by FSSP is huge. Looking at the full problem, one can notice that it includes actions that have nothing to do with achieving the goal.

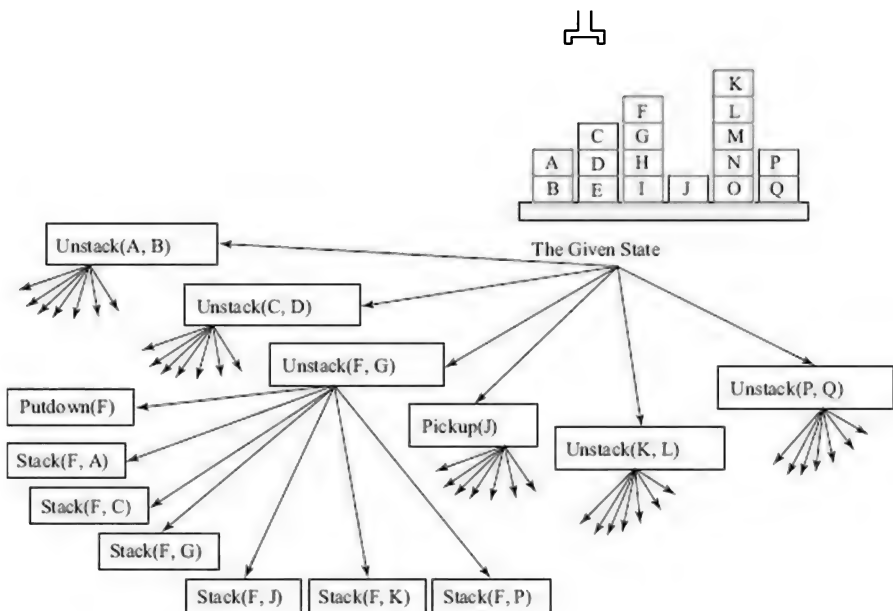


Figure 7.4 The search space generated by FSSP is huge.

7.3 Backwards State Space Planning

Like the uninformed search algorithms seen earlier, FSSP is oblivious of the goal, and simply explores the set of all future states possible in some

predetermined order. One way to take the goal state into account is to start searching backwards from the goal. That would make the reasoning purposeful and deliberative, focused towards the goal, or teleological. This is known as *Backward State Space Planning (BSSP)*. Backward reasoning or goal-directed reasoning has been proposed as being fundamental to intelligent behaviour, and we have already visited it while studying goal trees in Chapter 6. It is also the foundation of reasoning with logic, and the backbone of the logic programming language Prolog (Kowalski, 1974; Colmerauer, 1992). We will explore logic and reasoning later in Chapter 12. At this moment, let us focus on the possibility of backward state space planning.

One feature strongly in support of such an approach is the fact that goal states are often incompletely specified. That is, one often expresses only what is desired in the final state, rather than a complete description of the final state. For example, at some point of time, one might have the goal of satisfying hunger. Reasoning in a backward fashion, one will only focus on actions that will result in eating something, without worrying about anything else. Similarly, the goal in the planning problem of Figure 7.2 is that block *G* should be on block *A*, and block *B* on block *J*. Can we focus only on the moves that will achieve these conditions? To do this, we have to first define the notion of regression. This, in some sense, is the opposite of progression. It allows us to move back from a set of goal clauses to a set of subgoal clauses. For example, if in a regressed state the robot were holding block *G*, and block *A* was clear, and block *B* was on block *J*, then the *stack(G, A)* action would achieve the goal state of Figure 7.2.

Given a goal *g*, we say that an action *a* is *relevant* to *g* iff⁴

$$(g \cap \text{effects}(a) \neq \varnothing) \gg (g \cap \text{effects}^-(a) = \varnothing)$$

Let *preconditions(a)* denote the preconditions of action *a*. Given a goal *g*, and a relevant action *a*, one can regress to the goal *g'* as follows.

$$g' \leftarrow \{g - \text{effects}(a)\} \cup \text{preconditions}(a)$$

That is, from the set of goal facts, remove the effects of the action, and add the preconditions of the action. Let us define a function *regress(A, G)* that returns the regressed goal over action *A* when applied to goal *G*. The regressed goal represents the minimum set of facts that must be true in a state in which the action can be applied and which would result in a goal state.

However, the regression process is not sound, in the sense that the resulting set of predicates may not necessarily represent a state. This can happen because some actions that are relevant according to our definition may not be feasible in practice. This may result in an invalid plan. Figure 7.5 depicts some of the problems that may crop up in backward search. The algorithm begins with the given goal description *on(G, A) ∧ on(B, J)* for the problem depicted in Figure 7.2. Observe that if a goal predicate is true in the current state then one does not need to look for a relevant action. Two actions are relevant here, *stack(G, A)* and *stack(B, J)*. The former regresses to a goal *holding(G) ∧ clear(A) ∧ on(B, J)*, from which four operators are

relevant, as shown in the figure.

The first, *pickup*(*G*), would need block *G* to be on the table. But since the algorithm is working with incomplete descriptions, it does not know whether this will be so. If block *G* were on some block then the way to end up *holding*(*G*) would be to unstack it from some block, but which one? One way would be to instantiate an instance of *unstack* for each possible block in the problem. One can see that this will soon lead to an explosion in the search tree. The preferred method is to keep it as a variable, and instantiate it later. Some action to achieve *clear*(*A*) would be needed if block *A* were not already clear. And again, the algorithm does not know what should be unstacked from *A* to achieve *clear*(*A*). Furthermore, in the action *unstack*(?*X*, *A*) required for *clear*(*A*), the variable ?*X* can be bound only to *G*. This is because *unstack*(?*X*, *A*) also results in *holding*(?*X*), and the goal set requires *holding*(*G*) to be true. However, this connection between the requirements of the two predicates *holding*(*G*) and *clear*(*A*) requires some more sophisticated reasoning. Perhaps some kind of consistency check can force the binding of ?*X* to *G*.

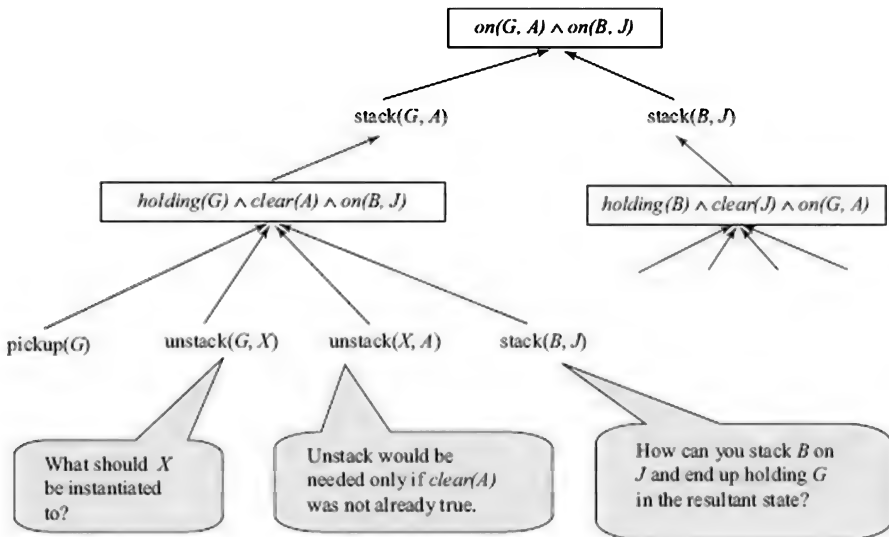


Figure 7.5 BSSP does not have a concrete state description to work with!

The last action *stack*(*B*, *J*) causes a more serious problem. It is meant to achieve the fact *on*(*B*, *J*). But one can notice that the robot arm could not be holding block *G* (asserted in the given goal) and block *B* (required for the stack action) at the same time. This shows that the algorithm *could be* reasoning with collections of facts that are not states. Figure 7.6 illustrates how a sequence of actions that is not a plan could be found by unhindered backward search. We consider a simple problem where in the given state three blocks, *A*, *B* and *C*, are on the table, and the goal is to stack *A* on *B*, and *B* on *C*. The figure shows a possible trace that backward search, using the regression process, might result in. The reader is encouraged to verify that at each stage of backward search the action chosen is relevant.

The reason why progression over states with planning operators works while regression does not, is that the operators are not symmetrical. They are designed to represent actions. An action converts a given state into a successor state. The applicability of the action is predicated only on the given state. If the given state satisfies the preconditions of the actions, it can be applied. Operators (or actions) have an in-built arrow of time, and the FSSP algorithm is aligned with it. Actions have preconditions on the given state which determine their applicability. The preconditions are necessary and sufficient for applicability. Once an action is applied to a given state, it produces a new state by adding some facts and deleting some facts. The point is that after progressing over an action, we necessarily land up in a state. One can say that the progression operation is *closed over the state space*.

Regression is done over sets of goal predicates. A goal represents facts that one wants true in some future state⁵. The relevance of an action to a goal simply means that *it looks like* that the action could achieve some part of the goal. But it does not mean that the action may be *applicable* in the preceding state. For example, for the goal in the above problem (Figure 7.6), the action *stack(B, C)* satisfies our definition of being relevant. But it obviously cannot be applied as a last action in a plan, because one can only move one block at a time, and block *A* needs to be on block *B*. The backward search algorithm being discussed could easily have found another invalid plan with *stack(B, C)* as the last action. While the goal description presumably applies to a state, one can observe that the regression operator is not closed over the state space. One may start with a (subset of a) state but end up with a set of predicates that is not the subset of any state. For example, the description "*holding(A) ∧ clear(B) ∧ holding(B) ∧ clear(C)*" in the figure cannot describe a state, because our robot has only one arm and cannot hold two blocks.

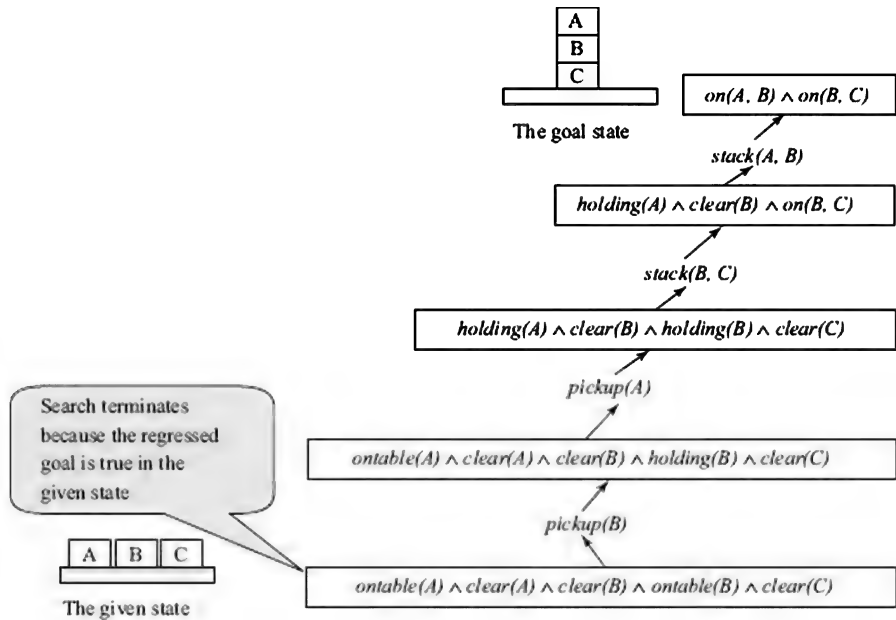


Figure 7.6 The plan $pickup(B), pickup(A), stack(B, C), stack(A, B)$ is not a valid plan.

Since we can have action sequences that are not feasible plans, let us define the notion of a valid plan.

Definition

A planning problem P is defined as a triple (S, G, O) where,

- S is set of facts completely describing the given or start state,
- G is a set of facts required to be true in a goal state, and
- O is the set of operators.

The above defines the simplest of planning problems. The simplest notion of a plan is a sequence of actions, each of which is an instance of some operator in O .

Definition

A sequence of actions $\Pi = (a_1, a_2, \dots, a_n)$ is a *valid plan* for a problem P , if a_1 is applicable in the state S , progressing to state S_1 , a_2 is applicable in the resulting state S_1 , and so on, yielding the state S_n after the action a_n , and $G \subseteq S_n$.

A simple linear time procedure may be written to verify that a plan $\Pi = (a_1, a_2, \dots, a_n)$ is a valid plan, by successively progressing through the sequence of actions and checking whether the final state contains the goal facts G .

```

ValidPlan(plan, state, goal)
1  if Empty(plan)
2      then if Satisfies(state, goal)
3          then return TRUE
4          else return FALSE
5      else state ← Progress(Head(plan), state)
6  return ValidPlan(Tail(plan), state, goal)

```

Figure 7.7 Algorithm to verify a plan. Function *Satisfies* checks whether the state satisfies the goal (that is $G \subseteq S$). Function *Progress* progresses the state over an action.

Given the function to check the validity of a plan, we can write a backward state space planning algorithm that will apply this check before termination.

In the algorithm described below, we also *assume* a function *Consistent*(*G*) that takes a goal *G* and checks whether *G* can be part of a state. Observe that this check does not have to be perfect. We use the check to filter out some inconsistent goals. The more inconsistent goals we can filter out, the faster our search algorithm will perform. But for the sake of correctness, the check should not filter out any consistent goals. The trivial correct case is when *Consistent*(*Goal*) always returns *true*. In this case, it does not filter out any goals, which means the algorithm may waste time over an infeasible sequence of actions. Some work on checking for consistency has been described in (Fox and Long, 1998; Kumashi and Khemani, 2002).

In the algorithm below, the search node is a pair made up of the goal to be satisfied, and the current plan. We use two functions, *Goal*(*node*) and *Plan*(*node*), to extract the goal component and the plan component from a given search node. They return respectively the first member and the second member of the list representing the node. The plan grows in a backward fashion, with the last action being found first. The planning algorithm takes as input the given state, the goal description, and a set of actions that are all possible ground instances of the set of operators for the given domain. One can adapt the algorithm to work with the set of operators itself. Working with instances is simpler for a finite domain. Also, not shown in the algorithm, is the role of a CLOSED list to check for looping. This is left as an exercise.

7.4 Goal Stack Planning

The main problem with BSSP is that goal regression is not sound and the algorithm may be handling goal descriptions that are not consistent with any state. The advantage of BSSP over FSSP is that because it focuses on the goal, the search space it generates is smaller due to the branching factor being lower. FSSP, on the other hand, is sound but has a large branching factor, since it looks at modifications to the complete state.

One of the earliest planners *STRIPS* (Fikes, 1971) attempted to combine the focused search of BSSP with the soundness of FSSP. It considered actions by reasoning in a backward manner, but committed itself to actions only in a forward manner. The algorithm is also called *Goal Stack Planning* (GSP) (Rich and Knight, 1990) because it pushes subgoals and actions into a stack, and picks an action only when all its preconditions are satisfied. That

way, it works with state descriptions that are always consistent for growing plans, and goal descriptions for growing the search tree. The algorithm is however incomplete, in the sense that it could terminate without finding a plan. It is possible that the algorithm could commit to a wrong action at some time, reaching a state from which a plan cannot be found. For domains where the actions are reversible, for example the blocks world domain, the algorithm can be made complete by putting in an extra check, somewhat similar to the one in BSSP. However, the reasons for introducing the check are different, as described below.

```

BSSP(givenState, givenGoal, actions)
1  goalPlan ← (givenGoal, ()) /* start with the empty plan */
2  open ← (goalPlan) /* note the extra set of brackets */
3  while not Empty(open)
4    do n ← Head(open) /*initially n = (GivenGoal, ()) */
5    goal ← Goal(n) /* extracts goal from search node n */
6    plan ← Plan(n) /* extracts plan from search node n */
7    open ← Tail(open) /* retain other search nodes */
8    if not Satisfies(givenState, goal)
9      then /* search further */
10         R ← set of relevant actions for goal /*may be empty */
11         for each action a ∈ R
12           do p ← Cons(a, plan)
13              g ← Regress(a, goal)
14              if Consistent(g) /* do not add if not consistent */
15                then
16                  gp ← List(p, g)
17                  open ← Cons(gp, open) /* Depth First */
18       else if ValidPlan(plan, givenState, givenGoal)
19         then return plan
20 return "No plan found"

```

Figure 7.8 Backward State Space Search regresses over goals. It validates a plan before returning it. It also assumes a function *Consistent* to prune away inconsistent subgoals.

GSP breaks up a set of goal predicates into individual *subgoals* and attempts to solve them *individually one after another*. This approach is also sometimes called *linear planning*. This refers to the fact that the subgoals are attempted and solved in a linear order⁶. When we implement BSSP with depth first strategy, we are in fact doing something similar. As we will see, this does not always work. In some domains, a subgoal solved earlier may get disrupted by later actions. We saw this depicted in Figure 3.15 for the Rubik's cube. Most human solvers of the cube attempt it as a sequence of subgoals, but end up disrupting them on the way to the final solution. In domains where the goal state is reachable from all states, one can tackle this problem of *subgoal interaction* by putting in a check to verify that all subgoals have indeed been solved before termination. This check is necessary to take care of subgoal interaction, which is different from the check in BSSP, where it was to determine the validity of the plan constructed.

Goal Stack Planning works by pushing the goal description onto a stack. It pushes both the conjunct, as well as each of the individual goal predicates separately. The algorithm pops the element on the top of the stack. If it is a (goal) predicate that is true in the current state then nothing is done and the next element is popped from the stack. If it is a goal predicate that is not true in the current state, a relevant action is pushed onto the stack, followed by

the preconditions—first the conjunction, and then the individual preconditions. The precondition on the top of the stack becomes the next subgoal to be addressed recursively.

In practice, one may have to choose from a set of relevant actions that is pushed onto the stack. In the algorithm described below, this is shown as a nondeterministic choice by a *choose* operator. In practice, during implementation, this should be a choice point for a backtracking search algorithm which will try out different actions.

If all the preconditions of an action are true in a given state, they will get popped away, and the action will be on the top of the stack. In this case, the action is popped and added to the partial plan being grown in a forward manner. The current state progresses over the chosen action, and planning resumes by looking at the top of the stack. If the stack becomes empty, all the goal predicates have been achieved, and a plan is returned. If at any stage the planner cannot find an action to achieve a goal predicate, the algorithm reports failure.

We assume a function *PushSet(Set, Stack)* that takes a set of goal predicates and (a) pushes the conjunction of the predicates onto the Stack, and (b) also the individual goal predicates in some order. This order may be determined heuristically if possible. For example, in the blocks world domain, *holding(Block)* should be attempted last because it blocks up the robot arm. If it is solved or achieved first then it could be undone while achieving some other subgoal, since the plan is for a one-armed robot. The algorithm for GSP is shown in Figure 7.9. We assume, as is the case with many modern planners, that all instances of the planning operators that match the domain are available, in a set called *actions*.

```
GSP(givenState, givenGoal, actions)
1  state ← givenState
2  plan ← ()                                /* start with the empty plan */
3  stack ← emptyStack                       /* start with the empty stack */
4  PushSet(givenGoal, stack)
5  while not Empty(stack)
6    do x ← Pop(stack)
7    if x ∈ actions
8      then plan ← (plan . x)               /* operator . extends plan with x */
9         state ← Progress(x, state)
10     else if x is a conjunct of goal predicates C
11       then solvedFlag ← TRUE
12          for each g ∈ C
13            do if g ⊆ state
14              then solvedFlag ← FALSE
15          if solvedFlag = FALSE
16            then PushSet(C, stack)
17     else if x ∉ givenState                /* x is an unsatisfied goal predicate */
18       then CHOOSE action a that achieves x
19          if no such action exists
20            then return FAILURE
21          Push(a, stack)
22          PushSet(Preconditions(a), stack)
23  return plan
```

Figure 7.9 Goal Stack Planning uses a stack to keep pending goals and actions. It picks actions only when it is applicable in the given state, and progresses forward.

Let us look at the trace of algorithm *GSP* on a small problem depicted in Figure 7.6. Let the initial state be:

$$S = \{\text{onTable}(A), \text{onTable}(B), \text{onTable}(C), \text{clear}(A), \text{clear}(B), \text{clear}(C), \text{armempty}\}$$

Let the final state be a tower of *A*, *B* and *C* expressed by $\{\text{on}(A, B), \text{on}(B, C)\}$. Assume that we execute the algorithm *GSP* and keep a watch on the *push* and the *pop* actions, the plan being assembled, and the current state *S* as and when it changes. Remember, that *PushSet* is a function that pushes the set onto the stack, first the conjunct of predicates, and then each one individually. Let us assume that the trace shows us the object being pushed or popped, the plan whenever it is augmented, and the state after the most recent action is popped. The trace of the algorithm will then look like the following:

```

S = {onTable(A), onTable(B), onTable(C), clear(A), clear(B), clear(C),
armempty}
Plan = ( )
  push → on(A, B) ∧ on(B, C)
  push → on(A, B)      /* choice: tackle on(B, C) first and then on(A, B) */
  push → on(B, C)
  pop  ← on(B, C)
  push → stack(B, C) /* an action to achieve on(B, C) */
  push → clear(C) ∧ holding(B) /* preconditions of stack(B, C) */
  push → holding(B)
  push → clear(C)
  pop  ← clear(C)      /* true in the given (start) state */
  pop  ← holding(B)
  push → pickup(B)     /* this is a choice point */
  push → onTable(B) ∧ armempty ∧ clear(B)
  push → armempty
  push → onTable(B)
  push → clear(B)
  pop  ← clear(B)
  pop  ← onTable(B)
  pop  ← armempty
  pop  ← onTable(B) ∧ armempty ∧ clear(B)
  pop  ← pickup(B)
S = {onTable(A), onTable(C), clear(A), clear(B), clear(C), holding(B)}
Plan = (pickup(B))
  pop  ← clear(C) ∧ holding(B)
  pop  ← stack(B, C)
S = {onTable(A), onTable(C), clear(A), clear(B), on(B, C), armempty}
Plan = (pickup(B), stack(B, C))
  pop  ← on(A, B)
  push → stack(A, B)
  push → clear(B) ∧ holding(A)
  push → holding(A)
  push → clear(B)
  pop  ← clear(B)
  pop  ← holding(A)
  push → pickup(A)     /* this is a choice point */
  push → onTable(A) ∧ armempty ∧ clear(A)
  push → armempty

```

```

push → clear(A)
push → onTable(A)
pop  ← cnTable(A)
pop  ← clear(A)
pop  ← armempty
pop  ← cnTable(A) ∧ armempty ∧ clear(A)
pop  ← pickup(A)
S = {cnTable(C), clear(A), clear(B), on(B, C), holding(A)}
Plan = (pickup(B), stack(B, C), pickup(A))
pop  ← clear(B) ∧ holding(A)
pop  ← stack(A, B)
S = {cnTable(C), clear(A), on(B, C), on(A, B), armempty}
Plan = (pickup(B), stack(B, C), pickup(A), stack(A, B))
pop  ← cn(A, B) ∧ cn(B, C)
Stackempty
Return Plan = (pickup(B), stack(B, C), pickup(A), stack(A, B))

```

The reader would have noticed that the algorithm has made some choices in the above execution. Choice points occur when there is more than one way of achieving a goal predicate. For example, the above trace has to find actions to achieve the goal predicate *holding(B)*. This can be achieved either by picking up block *B* from the table, or by unstacking it from some other block. Which block to unstack it from, would also be a choice point. Likewise, if the robot is holding a block, and the goal predicate being addressed is *armempty*, it has to decide where to put that block. One approach would be to introduce backtracking here. Another would be to do some secondary reasoning to resolve this choice. For example, one could look at the state to see which action would be appropriate. In the example trace, this would tell us that since block *B* is on the table, one can assume that nothing will disturb it till the time comes for *holding(B)* to be made true, the correct action is *pickup(B)*. The choice can also be resolved by looking at the goal set. If, for example, the robot is holding a block (say) *M*, and the goal has a predicate *on(M, N)* then a good way to achieve *armempty* would be to use the action *stack(M, N)*.

A different kind of choice is concerned with the order of attempting subgoals. By pushing the subgoals onto the stack one by one, the algorithm is serializing them. The order in which you attempt subgoals is obviously going to affect performance. In the given trace, the algorithm chose to solve for *on(B, C)* first. This turned out to be a good choice, and after stacking *B* on *C*, the algorithm stacked *A* on *B*. What if the order had been reversed? The algorithm would have first stacked *A* on *B* to reach the state,

$$S = \{\text{onTable}(B), \text{onTable}(C), \text{clear}(A), \text{clear}(C), \text{on}(A, B), \text{armempty}\}.$$

Then to achieve *on(B, C)* it would need to stack *B* on *C*, for which it would need *holding(B)*, for which it would have to *pickup(B)*, for which it would have to achieve *clear(B)*, for which it would have to *unstack(A, B)* and put it somewhere. It would have thus *undone the subgoal on(A, B) achieved earlier*. We have addressed this problem by adding the full goal conjunct in

the stack. After achieving $on(B, C)$ the algorithm will reach the state $S = \{onTable(A), onTable(C), clear(A), clear(B), on(B, C), armempty\}$ and again look at the conjunct $on(A, B) \wedge on(B, C)$. At this point $on(B, C)$ is true but not $on(A, B)$. Now it will pickup A and stack it on B to find the plan (pickup(A), stack(A, B), unstack(A, B) putdown(A), pickup(B), stack(B, C), pickup(A), stack(A, B)). It did solve the problem, but found a suboptimal plan.

Another place that ordering is imposed is in the order in which the preconditions are pushed onto the stack. For the action $pickup(?X)$ the preconditions are $onTable(?X)$, $clear(?X)$ and $armempty$. It makes sense to push $armempty$ first, and hence tackle it later. If $armempty$ is addressed first then the second precondition will need to disrupt it because the robot arm has to be used anyway. If for example $clear(?X)$ is done last then that would be achieved by unstacking something from $?X$, which would mean $holding(?X)$ would be *true* and $armempty$ would have got disrupted. Likewise, the preconditions of $stack(?X, ?Y)$ are $holding(?X)$ and $clear(?Y)$. For a similar reason $holding(?X)$ should be done later, and hence pushed first.

While in the above problem it was possible to find a subgoal ordering to solve them linearly, this may not always be the case. A seemingly similar problem (see Figure 7.3) known as the Sussman's Anomaly, demonstrates that even in simple problems in the blocks world, the subgoals may not be serializable (Sussman, 1975). That is, there is no ordering of subgoals where when each of the subgoals is solved individually, the goal is solved as a consequence. The progress of the GSP on Sussman's Anomaly is illustrated in Figure 7.10. We can observe that neither serialization of the two predicates will result in success. In both cases, the goal predicate achieved later undoes the one solved earlier. We have tackled this problem by adding an extra check for the entire conjunct. The algorithm will, therefore, realize that the goal has not been achieved and attempt the problem again, from the new state it is in. From here, it will find a solution in both cases. However, with this extra check we have created another problem. If the goal is inconsistent then the algorithm may never terminate. For example, if the goal were to be $on(A, B) \wedge on(B, C) \wedge on(C, A)$, the algorithm will go into an infinite loop.

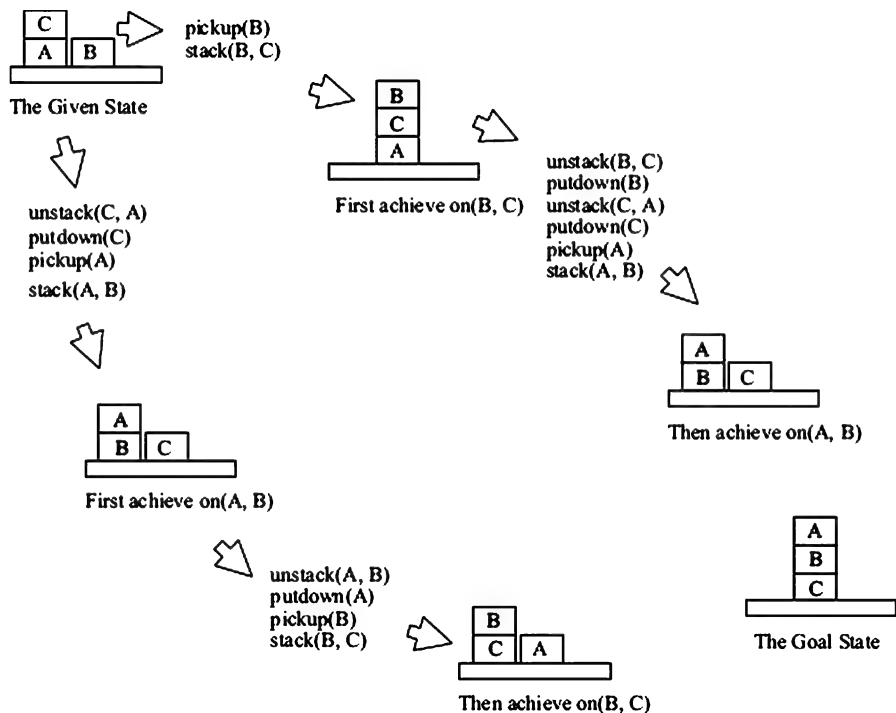


Figure 7.10 Neither order in Sussman's Anomaly solves the problem.

In the blocks world, the extra check works. This is because in this domain, any state is reachable from any other state. If this were not the case then the GSP algorithm would have run into problems. Consider, for example, planning in the culinary domain, where the actions are mostly irreversible. A planner, for example, may need to do some actions in a particular order. If it commits to a wrong set of actions then it would need to be able to go back to the choice point and choose another action. Thus for completeness, backtracking in the backward phase would be necessary.

We can think of the GSP algorithm as doing depth first search on an AND/OR goal tree. Pushing into the stack is like generating the tree. A goal predicate is *solved* if it occurs in the current state. An action node (AND node) is solved if all its children (preconditions of the action) are solved. An OR node represents a choice of actions for achieving a goal predicate. We will revisit this algorithm in the chapter on reasoning with logic, as backward chaining, the foundation of the programming language Prolog (see Chapter 12). The theorem proving algorithm searches for supporting statements in the backward direction, but commits to inferences in the forward direction. A recursive version (Figure 7.11) of the GSP algorithm adapted from (Ghallab et al., 2004) illustrates the dual nature of GSP search, and its relation to an AND/OR tree. It uses a function *PlanProgress(P, S)*, shown in Figure 7.12, that progresses a given state *S* over a plan *P* by successively progressing over its actions.

7.5 Plan Space Planning

The planning approaches described above reason with states. The planner is basically looking at a state and a goal. If the state satisfies the goal then it terminates. Otherwise, it makes a search move *over the state space* looking for actions to add to the plan.

An alternative view is to consider the *space of all possible plans*, and search in this space for a plan. We will call such approaches as *plan space planning*. Most algorithms in this category represent a plan as actions arranged in a partial order, and hence we also use the term *partial order planning*. Unlike in the state space methods described above, there is no restriction on the order in which actions are *added* to the plan. Since state space methods focus on the state, or on the goal which is a partial state description, the search methods look at ways to go to neighbouring states that are one move away. Thus, states are explored in a linear fashion, and the plans, which are themselves linear structures, grow linearly, being extended at one of the two ends. Plan space planning approaches work with plan structures, and have the ability, in principle, to modify or extend any part of a plan. That is why these methods are also known as *nonlinear planning* methods. Because of the fact that they can modify any part of a plan, the plan space planning approaches are not constrained to focus on any one subgoal continuously. They can shift attention midway, and in the process often solve problems, like the Sussman anomaly correctly as shown in Figure 7.13.

The start node for search in plan space planning is the empty plan Π_0 . It is represented by two actions that we will call A_0 and A_∞ . These two special actions will be part of every plan. The first, A_0 , has no preconditions, and its effects are the predicates describing the start state. The second, A_∞ , has as its preconditions the goal predicates, and has no effects, as shown in Figure 7.14 (for the tiny planning problem from Figure 7.18). Thus, every planning problem will have a distinct start node that will capture both the start state and the goal description.

```

RecursiveGSP(givenState, givenGoal, actions)
1  state ← givenState
2  goal ← givenGoal
3  plan ← {}
4  while TRUE
5      do if Satisfies(state, goal)
6          then return plan
7          else Let R be set of relevant actions for goal
8              for each subgoal g ∈ goal      /* AND node */
9                  do
10                     CHOOSE an action a in R /* OR node */
11                     if no such action exists
12                         then return FAIL
13                     subPlan ← RecursiveGSP(state, Preconditions(a), actions)
14                             /* searches in the backward fashion */
15                     if subPlan = FAIL
16                         then return FAIL
17                     else /* a deterministic version will backtrack */
18                         state ← PlanProgress(subPlan, state)
19                             /* progresses State in the forward direction */
20                         state ← Progress(a, state)
21                             /* assembles plan in the forward direction */
22                         plan ← (plan . subPlan . a)

```

Figure 7.11 Recursive GSP illustrates the dual nature of GSP. It considers actions by their relevance, but selects them only on applicability. For simplicity, we write a nondeterministic version with a CHOOSE operator that makes the correct choice. The deterministic version may make a wrong choice, but backtrack to try again.

```

PlanProgress(plan, state)
1  if Empty(plan)
2      then return state
3  else state ← Progress(Head(plan), state)
4  return PlanProgress(Tail(plan), state)

```

Figure 7.12 Algorithm *PlanProgress* iteratively progresses over the actions in a plan.

As planning proceeds, more actions are added to the plan. Interestingly, there is no constraint on where actions should be added. State space planning algorithms grow linear, partial plans at one end. Plan space methods separate the tasks of selection of an action and its placement in the plan.

In addition to the set of actions, the plan representation contains links between actions. The links are of two types. The first, called *ordering links*, are used to capture ordering information where it is known. The initial plan for example, has a default link $(A_0 < A_\infty)_\infty$ to assert that the start action A_0 happens before the end action A_∞ . The second kind of link, first introduced in a system called NONLIN (Tate, 1977), is called a *causal link*. A causal link (A_i, P, A_j) between two actions A_i and A_j can be established when an affect P of action A_i is a precondition for action A_j . Action A_i is the *producer* of predicate P , and action A_j is the *consumer* of P . Figure 7.15 illustrates a causal link between the two actions *Pickup(A)* and *Stack(A, B)*.

When the link is *established*, it represents a commitment on the part of action A_i to support the precondition P for action A_j . Once established, the algorithm will have to ensure that the link is not disrupted or *clobbered*. If it is

clobbered during the planning process then it will have to be *declobbered*. A causal link (A_i, P, A_j) is said to have a *threat* if there exists another action A_t in the plan that potentially deletes P , that is, has $(\text{not } P)$ in its effects. One may even treat an action as a threat if it produces P , because it threatens to make action A_i redundant (McAllester and Rosenblitt, 1991; Kambhampati, 1993).

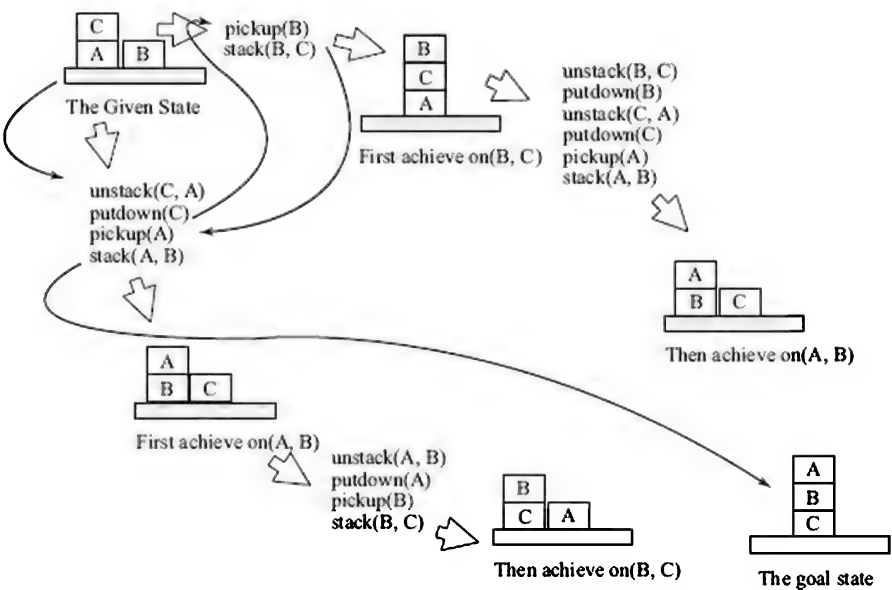


Figure 7.13 To find an optimal plan for the Sussman anomaly, the algorithm should start off with `on(A, B)`, switch to `on(B, C)` on the way, and return to `on(A, B)` again, instead of attempting them linearly.

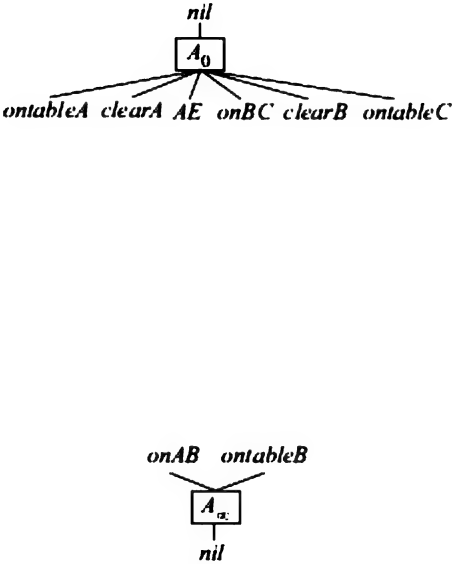


Figure 7.14 The empty plan Π_0 has two actions. Action A_0 produces the start state, while action A_∞ accepts the goal state.

If an action A has a precondition P that is not causally linked then we say that P is an *open precondition*⁷. A solution plan cannot have any open preconditions or threats. Together, the two are also called *flaws*. It has been shown (Penberthy, 1992) that if a partial plan does not have any flaws then it is a solution to the planning problem. The starting plan containing the two actions (A_0, A_∞) will be a solution plan only if the goal predicates are already true in the start state. That is, for each precondition of A_∞ , the action A_0 is the producer, and the corresponding causal links are established. If the algorithm can establish these links then it can terminate.

Plan space planning often follows what is known as a *least commitment strategy* (Weld, 1994). This implies that the algorithm commits to a particular feature in a plan, only when it has to. For example, in the blocks world, if the planner has to achieve *armempty* when it is holding a block say B . It might want to choose the *stack* action, but it may not know where to stack the block. It then makes sense to use only partially instantiated operators, and insert *stack*($B, ?X$)⁸ into the plan instead of choosing to guess and instantiate it to say *stack*(B, Q). We will use partially instantiated operators. Then for each variable, one has to keep track of the binding. So in addition to the two kinds of links, the plan representation will also contain a set of *binding constraints* that contain information of what specific variables can be bound to or cannot be bound to.

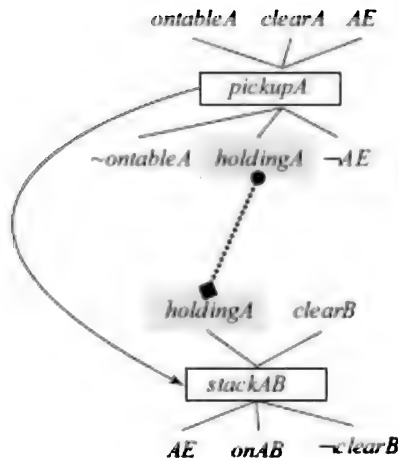


Figure 7.15 The causal link (*Pickup*(A), *holding*(A), *stack*(A, B)) is shown with the dotted link. The curved arrow is the ordering link between the two actions.

Thus, a node in plan space search will represent a partial plan, and will contain a set of partially instantiated operators, causal and ordering links, and binding constraints.

Definition

A partial plan P is a 4-tuple

$\Pi = (A, O, L, B)$ where

- A is the set of partially instantiated operators in the plan,
- O is the set of ordering relations of the form $(A_i < A_j)$,
- L is the set of causal links of the form (A_i, P, A_j) ,
- B is the set of binding constraints of the form $(?X = ?Y)$, $(?X \neq ?Y)$, or $(?X \in D_X)$ where D_X is a subset of the domain of $?X$.

The initial node in plan space search is defined by the partial plan,

$$\Pi_0 = (\{A_0, A_\infty\}, \{(A_0 < A_\infty)\}, \{\}, \{\})$$

The search space is the *implicit* directed graph whose vertices are partial plans and whose edges correspond to refinement operations. Each refinement operator transforms a partial plan Π into a successor partial plan Π' by augmenting one of the sets. The different refinement operations are,

- adding an action or partially instantiated operator to A ,
- adding an ordering constraint to O ,
- adding a causal link to L , or
- adding a binding constraint to B .

The choice in the refinement process is itself driven by the need to remove *flaws* from a partial plan. As described above, flaws can be of two types; open subgoals or threats. We look at ways to remove each of them.

An open goal refers to a precondition P of some action A_P in the plan that is not causally linked to another action. A causal link for P can be found in two ways.

1. If an existing action A_e produces P and it is consistent to add $(A_e < A_P)$ then one can establish a causal link (A_e, P, A_P) to the partial plan.
2. If no such existing action can be found then one has to insert a new action A_{new} to the partial plan, add the corresponding causal link (A_{new}, P, A_P) , and add the ordering link $(A_e < A_P)$ to the partial plan.

An action A_{threat} that can possibly disrupt an existing causal link (A, P, A_j) is a threat to the link. Disruption will occur if all three of the following happen:

1. A_{threat} has an effect $\neg Q$ such the P can be unified⁹ with Q .
2. It is possible for A_{threat} to happen after A_i .
3. It is possible for A_{threat} to happen before A_j .

If all the three happen then we say that the threat has materialized. To eliminate the threat, one needs to ensure that at least one of the three conditions for the threat is not met. This can be done by the following:

1. Separation Ensure that P and Q cannot unify. This can be done by adding an appropriate binding constraint to the set B in the partial plan.

2. Promotion Promote the action A_{threat} to happen before it can disrupt the causal link. That is, insert an ordering link $(A_{\text{threat}} < A_j)$ into the partial plan (add it to set O).

3. Demotion Demote the action to happen after both the causal link actions. That is, add an ordering link $(A < A_{\text{threat}})$ to the set O in the partial plan.

Figure 7.16 shows an example of a threat. Assume that at some stage there is a partial plan with two actions $pickup(A)$ and $stack(B, ?Y)$. Of the many open conditions, let the algorithm choose $clear(A)$ and insert a new action $unstack(?X, A)$ in the plan establishing the causal link $(unstack(?X, A), clear(A), pickup(A))$.

At this point, one can notice that the existing action $stack(B, ?Y)$ is a threat to the newly established causal link, because produces $\neg clear(?Y)$, and if $?Y$ is bound to A then it could *possibly* disrupt the link. To resolve this threat, one of the three methods described above can be chosen. They are given below:

1. Separation Ensure that $clear(?Y)$ and $clear(A)$ cannot unify. This can be done by adding the binding constraint $(?Y \neq A)$ to the set B in the partial plan.

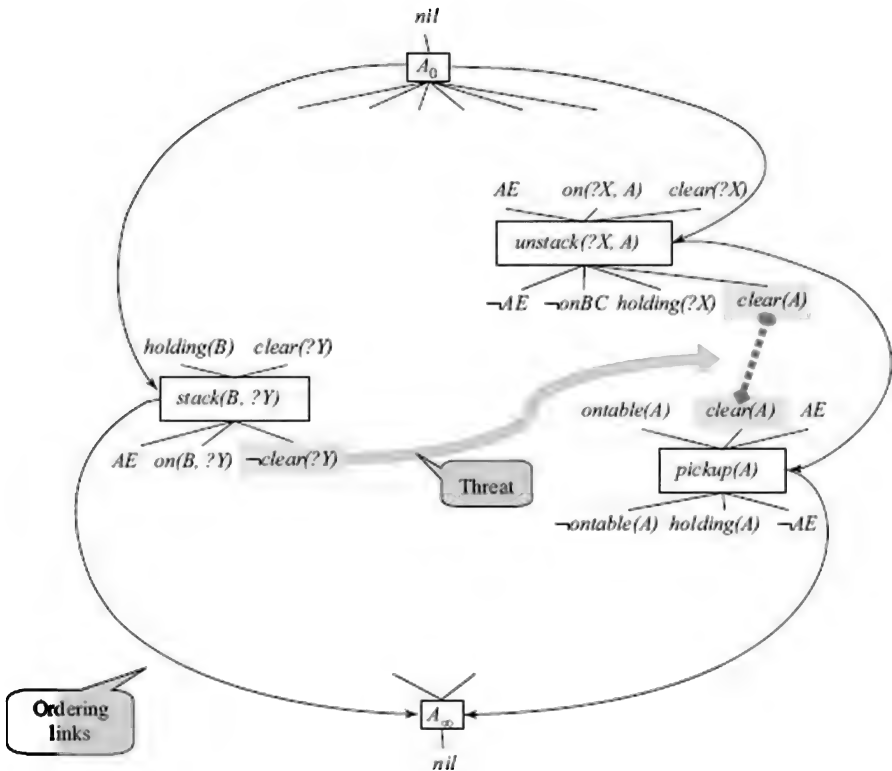


Figure 7.16 The action $stack(B, ?Y)$ is a threat to the causal link for the proposition $clear(A)$ produced by $unstack(?X, A)$ and consumed by $pickup(A)$.

2. Promotion Promote the action $stack(B, ?Y)$. Insert an ordering link $(stack(B, ?Y) < unstack(?X, A))$ into the partial plan.

3. Demotion Demote the action $stack(B, ?Y)$ to happen after the causal link actions. That is, add an ordering link $(pickup(A) < stack(B, ?Y))$ to the set O in the partial plan.

At any stage of planning, a refinement step can be applied to the partial plan, provided it does not introduce an inconsistency. For the ordering links to be consistent, there must be no cycles in the directed graph representing the ordering relations. In other words, the action must begin with A_0 and move forward without turning back, ending with A_∞ . Ordering links are added (a) along with causal links and (b) during promotion or demotion of actions. Before each of these operations, a check must be made for consistency. Likewise, for the binding constraints to be consistent, all assignments to variables should be consistent with all the binding constraints. Whenever a new binding constraint is added, a check for consistency for the variables involved must be made¹⁰. The causal links too must not have cycles, but a check is not required because whenever a causal link is added to the partial plan, a corresponding ordering link is added as well, and it suffices to check that the ordering links are consistent.

The basic plan space algorithm described below (Figure 7.17) is adapted from (Ghallab et al., 2004). It is initially invoked with the empty plan P_0 as the argument.

```

PSP( $\pi$ )
1   $flaws \leftarrow OpenGoals(\pi) \cup Threats(\pi)$ 
2  if Empty( $flaws$ )
3    then return  $\Pi$ 
4  CHOOSE  $f \in flaws$ 
5   $resolvers \leftarrow Resolve(f, \pi)$  /* the set of resolvers for flaw  $f$  */
6  if Empty( $resolvers$ )
7    then return FAIL
8  CHOOSE  $r \in resolvers$ 
9   $\pi' \leftarrow Refine(r, \pi)$ 
10 return PSP( $\pi'$ )

```

Figure 7.17 The plan space planning (PSP) procedure selects a flaw in a given plan, and looks for a resolver that can remove the flaw. The Refine procedure applies the chosen resolver, and the algorithm PSP is called recursively.

In Line 1, the set of flaws is identified as the union of the set of open conditions and the set of threats. The procedures that return the two constituent sets need to maintain the two sets and update them incrementally in each cycle during the call to *Refine* in Line 9. The set of open conditions is usually maintained as an agenda of goals, to which new ones are added every time open conditions are created. Likewise, every time an ordering condition is added, a check is made for the possibility of a threat. Observe that whenever actions are added, one or more causal links as well as ordering links are added as well. In Line 4, one of the flaws is chosen for resolving. In theory, any flaw could be chosen because all the flaws have to be resolved for the algorithm to terminate. In practice, while implementing a deterministic algorithm choosing a flaw that has a smaller number of resolvers could lead to less backtracking. In Line 5, the set of resolvers for the chosen flaw that can be consistently applied are identified. The algorithm has to check that no ordering constraints and binding constraints are violated by the addition on new ones. This ensures that in Line 9, the resolver, chosen nondeterministically¹¹ in Line 8, can be applied without any problem, and procedure *Refine* has only to update the relevant structures being

maintained. The *PSP* algorithm treats both kinds of flaws equally.

A variation of the above algorithm called *Partial Order Planner* (POP) works only with open goals in an agenda. Every time it finds a way of satisfying the open goal, it looks for any threats created and resolves them before moving onto to the next goal on the agenda. Thus, plan space planning or partial order planning can be seen to go through a cycle of “R” steps. Remove a flaw (or an open goal) from the agenda. Resolve the flaw. Refine the partial plan, and in the process, Revise the agenda.

Let us look at a small but complete example. Let the initial state of the problem be $\{ontable(A), clear(A), ontable(C), on(B, C), clear(B), AE\}$ where *AE* stands for *armempty*. Let the goal predicates be $\{on(A, B), ontable(B)\}$. The problem is depicted in Figure 7.18.

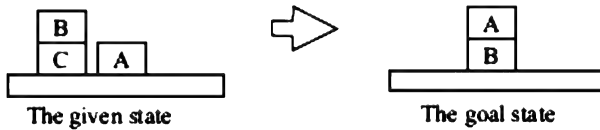


Figure 7.18 A small planning problem. Note that nothing is said about block C in the goal state. It could be on the table or on block A.

Figure 7.19 shows a partial plan after four actions have been added as described below.

The empty plan with actions A_0 and A_∞ captures the initial state and the goal clauses. The two goal clauses are the open conditions. The planner chooses one of them, say $on(A, B)$, and inserts an action $stack(A, B)$. It inserts an ordering link ($stack(A, B) < A_\infty$), and a causal link ($stack(A, B), on(A, B), A_\infty$). The preconditions of the action $stack(A, B)$ now appear as open conditions, $holding(A)$ and $clear(B)$.

Let us say that the planner then looks at $clear(B)$ and finds that it has a producer in the action A_0 . It establishes a causal link (shown in the figure) and an ordering link (not shown) for $clear(B)$. It could then take up the open condition $ontable(B)$ and add the action $putdown(B)$, and the corresponding links. Next comes the open condition $holding(A)$, and for that, it inserts the action $pickup(A)$ and its links.

The only open condition left at this point is $holding(B)$, and for the planner, insert the action $unstack(B, C)$ into the plan. At this point, three threats appear, marked as t_1 , t_2 and t_3 in the figure.

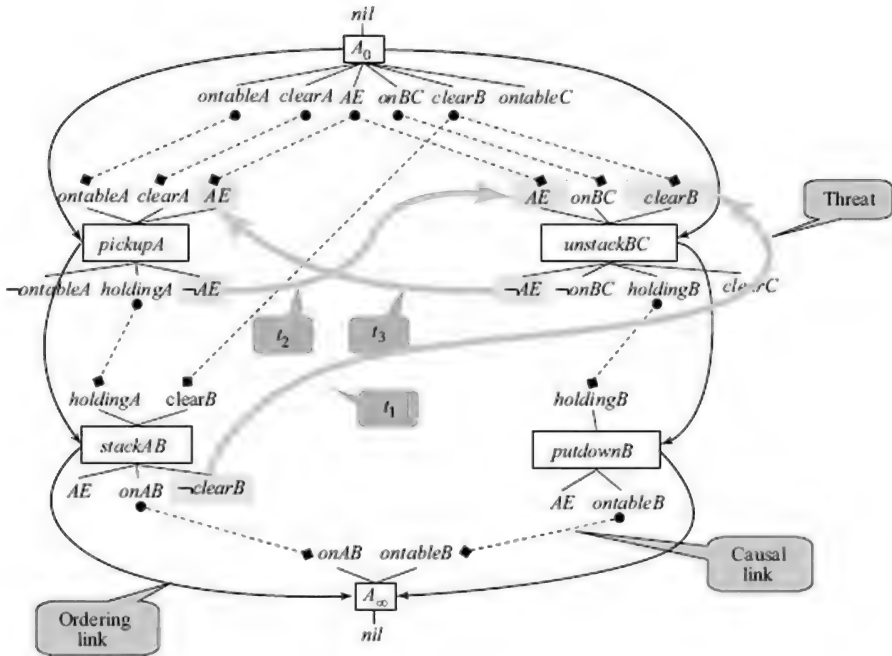


Figure 7.19 A partial plan with four actions added in the order *stackAB*, *putdownB*, *pickupA*, *unstackBC*. After the last action, there are no open conditions, but three threats *t1*, *t1*, and *t3* appear.

Threat *t1* is that if *stack(A, B)* is done before *unstack(B, C)* it will disrupt (or clobber) the precondition *clear(B)* of the latter. This can be resolved by demoting action *stack(A, B)* to happen after *unstack(B, C)*. This is done by adding an ordering link (*unstack(B, C) < stack(A, B)*) into the plan.

Threat *t2* and threat *t3* are symmetrical in nature. Let us assume that the planner correctly resolves *t2* by likewise demoting action *pickup(A)* by adding the link (*unstack(B, C) < pickup(A)*). Notice now that the threat *t3* has become definite, in the sense that we know that the AE condition for *pickup(A)* is going to be clobbered by *unstack(B, C)*. The corresponding causal link is broken, and we have the open condition AE for *pickup(A)*. We cannot demote *unstack(B, C)* without making the ordering links inconsistent. Instead, the planner looks for an action to achieve AE again. It finds it within the plan itself. Action *putdown(B)* can produce AE for *pickup(A)* to consume. Thus, declobbering, as it was first described in a system called TWEAK (Chapman, 1987), is done by inserting ordering link (*putdown(B) < pickup(A)*) and causal link (*putdown(B), AE, pickup(A)*) between the two actions. The resulting plan with no threats or open conditions is shown in Figure 7.20. Only the necessary subset of ordering links is shown for clarity, like in a Hasse diagram.

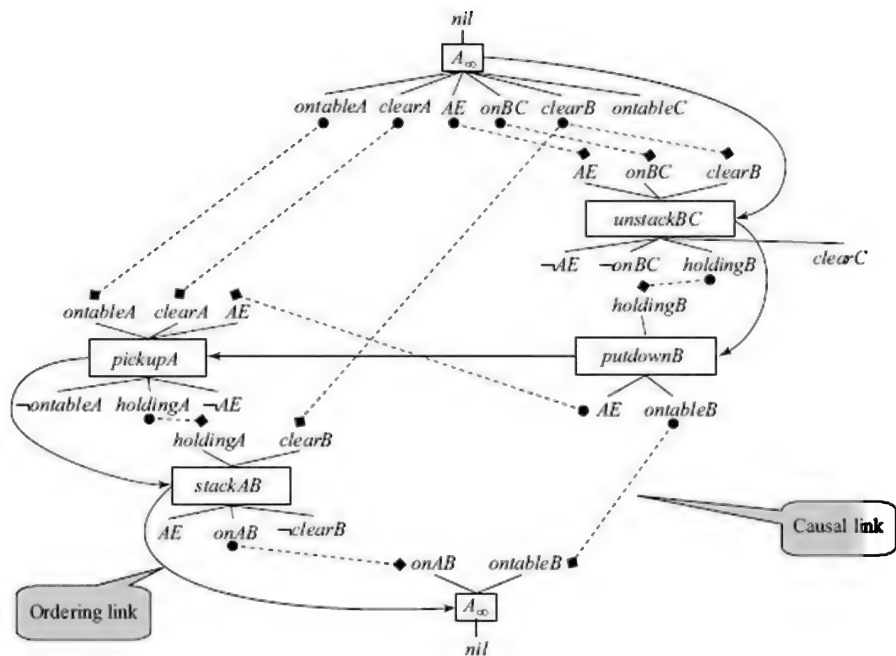


Figure 7.20 The final plan after threats t_1 , t_2 and t_3 are resolved. There are no open conditions or threats, and hence this is a solution plan.

The reader would have observed that the resulting plan is a linear plan. This is inevitable in the blocks world domain, because the one-armed robot can hold only one block, and therefore only one action can be done at a time. If the robot could hold more than one object then one could have partial plans that are not linear orders. For example, a parallel step could say *pickup(A)* and *pickup(B)* without specifying the order, that is there would be no ordering constraint between the two actions, and the plan could be correctly linearized in any order.

An example of such parallel actions could be the “dressing up for school” problem, for which the plan is shown in Figure 7.21. The subplan has four actions—*wear(left, sock)*, *wear(right, sock)*, *wear(right, shoe)* and *wear(left, shoe)*—that follow a *comb-hair* action with only constraints that each sock must be worn before the respective shoe. Otherwise, the actions could be done in any order. This gives us a compact representation of a plan, which stands for all the linear plans that are consistent with the ordering constraints.

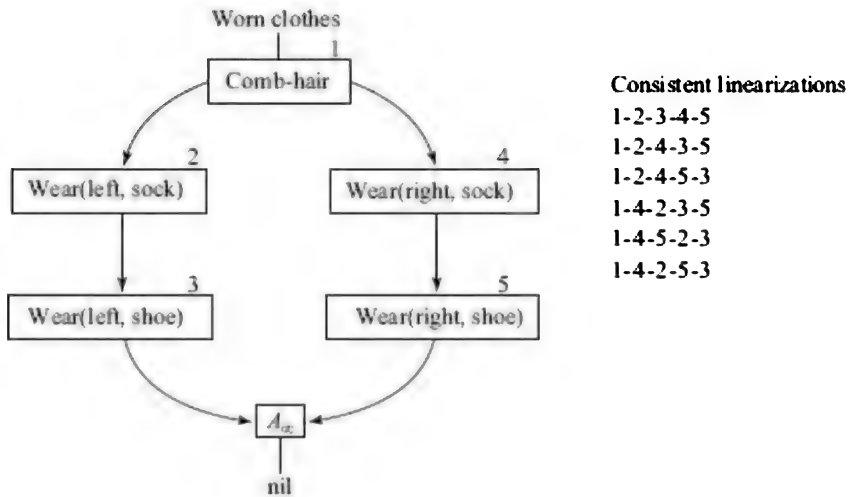


Figure 7.21 A partial plan may stand for many linear plans.

The algorithms above have been written in a nondeterministic manner. While this results in a succinct description, it hides the fact that the algorithm still has to search through a combination of choices. In practice, one will have to choose a combination of heuristics to guide the search, and introduce backtracking to make it complete. The reader might have noticed that the search space for PSP is infinite, even when the state space is finite. Given a two block problem in the blocks world domain, there are only four distinct states possible. But the number of candidate plans is infinite, because one can always insert the two actions (*pickup*(X), *putdown*(X)) where applicable *any number of times*. Since the representation does not involve states at all, one cannot check that the same state is being visited again and again. One can, in principle, explore plans of arbitrary length without moving towards a solution. Thus, there is a need to control the search strategy efficiently. One way to do so is to explore plans in an iterative deepening manner, keeping an iteratively increasing bound on the number of actions in the partial plan. This would ensure that plans are considered in the order of increasing length, and the solution plan would be found in finite time.

The reader would also have noticed that the PSP algorithm has a backward reasoning flavour, since it looks for ways to resolve open conditions, though the representation is very different from BSSP. While BSSP can run into spurious states, PSP avoids that pitfall having done away with states altogether in its representation. The sequence in which both consider actions can still be similar. The partial plan representation on the other hand is powerful enough to accommodate other search strategies, with some modifications as discussed below. The approaches described below have a flavour of top down or goal-directed problem solving.

7.5.1 Means Ends Analysis

Means Ends Analysis (MEA) is a strategy first introduced in a system called *General Problem Solver (GPS)* (Newell and Simon, 1963). It was one of the first general purpose approach to problem solving, or planning. The basic idea in *MEA* is to,

- Compare the given state and the desired state and arrive at a set of differences
- Choose the most significant difference and look for a plan to reduce that difference recursively
- Apply the operators chosen and look at the problem again for any more differences

We can think of the *GSP* algorithm in *STRIPS* as an example of the *MEA* strategy. As described in Figure 7.11, the *recursiveGSP* algorithm looks at the goal propositions, and attempts to solve one of them before considering the next one. The *STRIPS* planner focuses on the goal propositions. Each goal proposition is a difference to be tackled. It chooses a goal to tackle from the goal set in a predefined manner. *MEA*, on the other hand, has a more top-down view of problem solving, and it requires the following set of procedures,

- A *MATCH* procedure that compares two states and returns a set of differences, if any, between them. There exists an efficient algorithm for doing this, the *Rete* algorithm, described in Chapter 6 in a different context.
- A procedure for ordering the differences to arrive at a set of goals (Ends) to be achieved
- A set of operators (Means) to reduce the differences.

The versatility of the algorithm depends upon the availability of knowledge for the above three procedures. In *STRIPS*, the set of differences is simply the set of open goals or subgoals, and as observed above, it does not order the differences. The set of operators can be provided in a modular form as in *PDDL*. Consequently, the algorithm can be written in a domain independent form. On the flip side, as we have seen, *GSP* does not always find the best plans. In more general cases, some of the above knowledge would require more domain related procedures and may not be general enough. We illustrate the *MEA* strategy with a few examples below. The reader is encouraged to think of the strategy as working on a partially ordered plan structure, in which actions are inserted because they address the most significant differences between the desired state and the *effects* in the current partial plan. In particular we may insert actions based on criteria other than open condition satisfaction and threat removal, and work with a more general notion of flaws which may vary from domain to domain. We can think of the flaws (in the more general sense) as the differences in *GPS*. Furthermore, we order the flaws in some order of importance or difficulty and tackle them in that order.

Consider the task of planning a trip from IIT Madras (IITM) in Chennai to the Technical University in Munich (TUM). The problem is to overcome the difference (distance) in our current state (at IITM) and the desired state (at TUM). An *FSSP* may start by planning with applicable moves from IITM, and a *BSSP* or *GSP* may start looking at the relevant moves from the goal (at TUM). They would try and construct the plans in linear fashion at the level of

detail of actions. A *GPS* like solver may first address the biggest difference, in terms of distance, and observe that we can reduce the distance from Chennai to Frankfurt (the most significant difference) because we have the means to do so. That is, there is a flight from Chennai to Frankfurt. As one can notice, this information comes from the domain. We often think of such an operator first because we visualize the differences as distances and operators as means of transport. We could be working with an operator difference table that looks as follows.

Table 7.1 A possible operator difference table for travelling, lists the modes of transport that can be used to cover different distance ranges.

Distances	Modes of transport					
	Aeroplane	Train	Car	Taxi	Bus	Walk
More than 5000 km	yes					
100 km to 5000 km	yes	yes	yes			
1 km to 100 km		yes	yes	yes	yes	
Less than 2 km			yes		yes	yes

Observe that it is still not straightforward to use this table, because looking at the original problem, one has to choose from the set of available options and operator that would reduce the largest difference. Selecting a Chennai–Frankfurt flight¹², for example, implies the knowledge that there are no Chennai–Munich flights. Once we manage to decide upon the operator, we are left with two new problems to solve. They are,

1. Given: (at IITM), Desired: (at Chennai airport)
2. Given: (at Frankfurt airport), Desired: (at TUM)

One can solve these independently, consulting the operator difference table at each stage. A more concrete example that is easier to implement is the Towers of Hanoi¹³ problem. This problem

also illustrates the utility of being able to order the differences. The domain constitutes of three pegs, or locations, and a tower of N disks on one peg to start with. The disks are all of different diameters, and any disk can be placed only on top of a larger disk or on an empty peg. A move constitutes of transferring a disk from the top of one peg, to another peg where possible. Figure 7.22 depicts a problem with three disks.



Figure 7.22 The Towers of Hanoi. The task is to move the tower of N (3 in the figure) disks from location peg A to peg C. Only the topmost disk on a peg may be moved, and cannot be placed on a smaller disk. The problem is known to have solutions (plans) that are exponentially long: $(2^N - 1)$ moves for towers with N disks.

The Towers of Hanoi problem is very structured and often used to illustrate recursive algorithms (to move a complete tower move all disks, but

one recursively to a temporary peg, move the remaining disk to the destination, and move the other disks recursively to the destination). However, if the problem is treated as a planning problem, it poses a challenge because the optimal solution is known to be exponential in length. For a problem with N disks, the number of moves in the solution is $(2^N - 1)$. Combined with the fact that in each intermediate state, either two or three moves are possible, one can see that the search space is very large.

The *MEA* strategy works well with the ordering information that the largest disks are hardest to move since they may have more disks on top of them. Then, if the differences measured are in terms of differences in location of each disk, the differences for the largest disks must be reduced first. In Figure 7.2, three disks have to be moved from tower A to tower C. The differences in order of importance are,

- D_3 : disk d_3 is on peg A and not on peg C
- D_2 : disk d_2 is on peg A and not on peg C
- D_1 : disk d_1 is on peg A and not on peg C

Having chosen the difference D_3 to reduce, *GPS* creates a goal G_3 of reducing that difference. This is because the operator to reduce the difference may not be applicable in the given state. For example, the operator to reduce D_3 is to move disk d_3 from A to C, but that can be only done if disk d_3 is clear, and peg C is empty (because d_3 is the largest disk; otherwise the condition would be that the existing disk on peg C is larger). These conditions would be true if disks d_1 and d_2 were not on peg A, and also not on peg C. Thus, *GPS* would recursively create a new set of differences as follows,

- D_{32} : disk d_2 is on peg A and not on peg B
- D_{31} : disk d_1 is on peg A and not on peg B

And work towards reducing D_{32} . Continuing in the same manner, it would create a new goal to move d_2 from A to B, and then another to move d_1 from A to C. This is a move it can make, and it will do so going to a new state S_1 in which d_1 is on C. It can now move d_2 from peg A to peg B, thus achieving the goal of reducing D_{32} . Now it looks at the difference D_{31} and revises the difference to

- D'_{13} : disk d_1 is on peg C and not on peg B

which can be reduced by moving d_1 from peg C to peg B. Having done all this, *GPS* is now in a position to solve goal G_3 by moving disk d_3 to peg C. After this, it goes on to reduce the revised differences D_2 and D_1 , revised because the world may have changed in the interim period. The reader would recognize the similarity of the “trace” with the one for *GSP*, and in fact as observed earlier, *GSP* is a special case of *GPS*. If one were to use the *GSP* planner to solve the Towers of Hanoi and give it additional information on ordering of goal propositions (lowest blocks/disks in goal set first), we would have in effect the algorithm described above.

Problem decomposition by the *MEA* strategy can also be seen to generate an AND/OR tree (Nilsson, 1971). At the *AND* level, the strategy breaks up the problem into two parts, which are ordered (see Figure 7.23). In the first part, it poses a goal to reduce the largest difference, and in the remaining part, it addresses the remaining differences. Reducing the largest

difference itself could be done in several ways and each of them becomes a choice at the *OR* level. Having selected an operator, the *MEA* strategy recursively poses the problem of achieving the preconditions for that operator and the task of applying that operator when its preconditions are achieved.

The reader would again have noticed the likeness in structure of the *recursiveGSP* algorithm in Figure 7.11 and Figure 7.23 showing the AND/OR tree above. One of the many contributions of the pioneering work on *GPS* is the recursive approach to problem solving.

The key to performance in *GPS* is an ordering of the differences to be addressed, and the availability of operators to reduce the differences. One desirable property of the operators is that applying an operator to reduce a difference should not undo the work of a previous operator. We have seen that such a problem can occur even in the blocks world with the example on Sussman's anomaly. While it may not be possible to find primitive operators that satisfy this property, Korf has shown that one could construct operator difference tables in which the operators are not primitive operators, but chunked together into macros. In other words, one can even tackle nonserializable subgoals by devising macro moves that can be applied in a serial order (see also the section on Peak-to-Peak Heuristics in Chapter 3). Table 7.2 below shows the macro table for the Eight-puzzle (taken from (Korf, 85)) for the goal state shown in Figure 7.24.

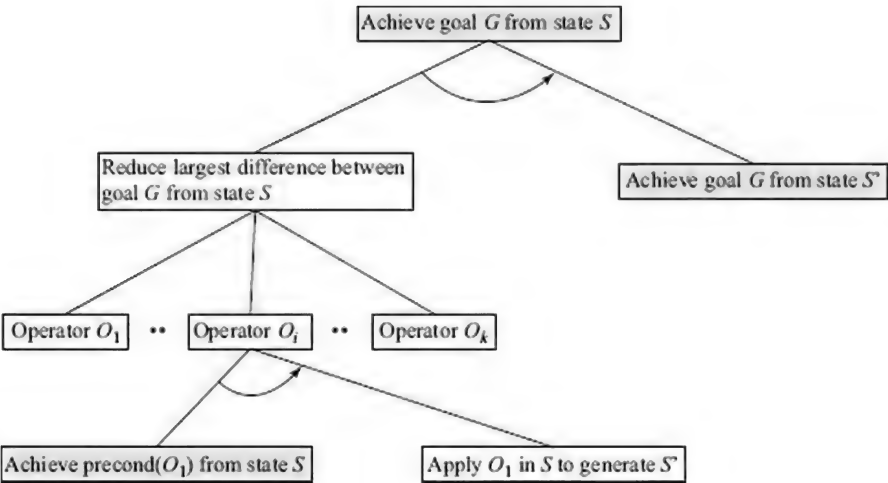


Figure 7.23 The *MEA* strategy generates an AND/OR tree by replicating the above structure below the shaded nodes when a recursive call is made.

Table 7.2 A macro operator table for the Eight-puzzle

		TILES						
		0	1	2	3	4	5	6
P O S I T I O N S	0	—						
	1	UL	—					
	2	U	RDLU	—				
	3	UR	DLURRDLU	DLUR	—			
	4	R	LDRURDLU	LDRU	RDLLURDR UL	—		
	5	DR	ULDRURDL DRUL	LURDLDRU	LDRULURD DLUR	LURD	—	
	6	D	URDLDRUL	ULDDR	URDDLULD RRUL	ULDR	RDLLUURD LDRUL	—
	7	DL	RULDDRUL	DRUULDRD LU	RULDRDLU LDRUL	URDLULDR	ULDRURDL LURD	URDL
	8	L	DRUL	RULLDDR	RDLULDRR UL	RULLDR	ULDRRULD LURD	RULD

An entry in row I and column J is a macro move for reducing the difference in the position of tile J if it is currently in the final position of tile I . Tile 0 stands for the empty tile. The macro moves are composed from four primitive moves R , D , L , and U which stand for moving an appropriate tile right, down, left or up. Note that the notation is unambiguous, because only one tile can make each of the primitive moves at any time. For example, in the position in Figure 7.24, move L means tile 4 is moved into the empty place, and U means tile 6 is moved.

The shaded squares can be interpreted as follows. If square at row I and column J is shaded, it means that the move to reduce the difference for tile J does not interfere with the location I (position of tile I in the goal state). In other words, the difference for tile I is invariant to the reduction of difference for tile J . In general, if we have operators such that we can arrange the invariance between differences in a triangular form then one can reduce the differences in a serial order (see note by G Ernst (1987) for another example of the triangle property for the Fool's Disk puzzle).



Figure 7.24 The Goal State for which the macro Table 7.2 is constructed.

In the above table, the differences are arranged in a triangular invariance order. This means that if we reduce the differences in the given order then the problem can be solved in a linear fashion. Thus, the differences to be reduced are in the order from tile 0 to tile 6, by the end of which the solution is reached. In other words, given any starting position, one can find a solution for the above goal in the Eight-puzzle by bringing tile 0 into place, bringing tile 1 into place, and so on using at each point the appropriate macro from the table. The reader would have observed that these are not the only macro moves that are possible. For example, the above table does not

tell us what to do if we want to reduce the difference for tile 4, if it is in the position of tile 2. The point is that such a macro is not required for completeness, because by the time the turn of tile 4 comes, tile 2 would already be in place, and hence tile 4 could not have been in the location of tile 2. Also the macros are not unique. For example, the macro UL at location (1, 0) could also been LU. It has been shown that if we can find operators that allow us to arrange the differences so that the invariance relations can be arranged in a triangle then the solver is complete (Banerji, 1977). Using the macro table, one can solve the problem from any starting position when a solution exists. The solution found may not be optimal though. Finally, one could choose a different order of solving the differences, and then find the appropriate macro moves. In fact, the work by Korf demonstrates how this can be done.

7.5.2 NOAH

An important characteristic of top-down problem solving is that one should start by looking at the problem in totality at the abstract level, and then work on the detail. The search based methods we have seen earlier operate at the level of primitive moves or operators, and try and synthesize plans or solutions from them. In the *GPS* formulation, one does look at a problem and address the largest difference, but it needs an appropriate set of operators. Korf showed that one can construct appropriate macros from primitive operators and use them in planning with the *MEA* strategy. But in doing so, all the hard work has been pushed into the task of finding or learning those macro operators. Given the appropriate operator difference table, there is no search required any longer, and the plan or solution can be constructed by looking up the table. There is a need for search algorithms to be able to operate at different levels of abstraction. *STRIPS* like systems also use the *MEA* strategy, but are compelled to work with the primitive operators from the word go.

One of the first systems to adopt a hierarchical approach to planning was *ABSTRIPS* (Sacerdoti, 1974). As the name suggests, *ABSTRIPS* is an extension or abstraction of *STRIPS* and works on the same problem representations. It, however, uses some additional information to search in a different manner. The additional information it uses is a partial ordering of the predicates in the domain, determined by a *criticality value* assigned to each predicate. This criticality value is very similar to the ordering of differences in *GPS*. *ABSTRIPS* start off by ignoring all but the most critical predicates and searches for a solution, with only the most critical predicates in the preconditions. During this process, it ignores all other preconditions of the operators. Having found a plan, it moves to the next level of criticality. This is done by looking at the predicate with the next level of criticality, and posing subproblems to achieve them wherever they become visible in the top level plan. The solutions for these subproblems are inserted into the top level plan, and the algorithm moves on to the next level.

The hierarchy of abstraction in *ABSTRIPS* is generated by ignoring preconditions in the decreasing order of criticality. Another way a hierarchy of

actions can be generated is by considering the effects of groups of actions, somewhat like the macros described earlier, but treated as operators. This would enable the problem solver to operate with higher level tasks. Thus, planning a vacation trip might have a high level solution like (book-tickets, go-to-railway-station, travel-to-destination, look-for-hotel, check-in-to-hotel, hike-for-three-days, return) at different levels of abstraction¹⁴. This solution will in turn have to be expanded into sequence of actions at the primitive action level. It is the set of primitive actions that can be carried out in the domain. Thus, high level operators are like instructions in high level programming languages that have to be interpreted by sequences of the primitive instructions provided by the machine hardware. One of the first systems to look at planning in such a hierarchical space was called *NOAH* (Networks of Action Hierarchies), also developed by Sacerdoti (Sacerdoti, 1977). *NOAH* represents plans as *task networks*. The task networks are partial orders, but are not partial plans. Instead, they represent complete plans at a high level of abstraction. *NOAH* solves a problem by successively refining parts of the task network, till the network has only primitive tasks that can be done by domain operators. After every refinement step, *NOAH* passes control to a set of routines called *critics*. Each critic inspects the partial plan for a particular defect and, if found, suggests a remedial action.

We look at how *NOAH* would solve the Sussman's Anomaly. *NOAH* starts off like *GPS* by addressing the problem at a high level and proceeds to work in the details, by refining a part of the task network. The initial task network contains a single node. The given task is to achieve the goal " $on(A, B) \wedge on(B, C)$ ". Like *STRIPS*, *NOAH* too decomposes this task into two subtasks for each of the subgoals, but unlike *STRIPS*, it does not impose an order on the two subtasks. Instead, as shown in Figure 7.25, it creates a network by adding two special action nodes called *split* and *join*, to form a partial order. This is the *least commitment strategy* in which the planner delays commitment to decisions, until it can make an informed choice. The plan at this level is to split the task into two subtasks, then solve each of them in some order, and finally join the results of the two subplans. In the next two steps, *NOAH* refines the two tasks, as shown in Figure 7.26 below. It replaces, or expands, each of the nodes into networks of lower level actions. The task "*Achieve on(A, B)*" can be accomplished by clearing blocks *A* and *B*, in some order, and then "putting" *A* on top of *B*. The higher level action "Put" can be expanded like a macro into *pickup* or *unstack*, as the case may be, followed by the *stack* operator.

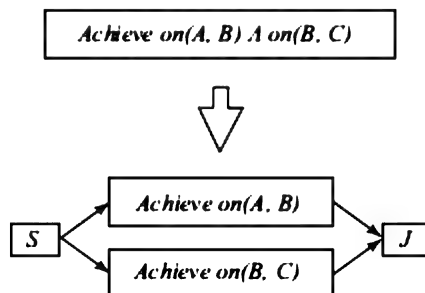


Figure 7.25 NOAH starts off by decomposing the higher level task into two tasks without committing to an order. The two tasks are linked by a Split node *S* and a Join node *J*.

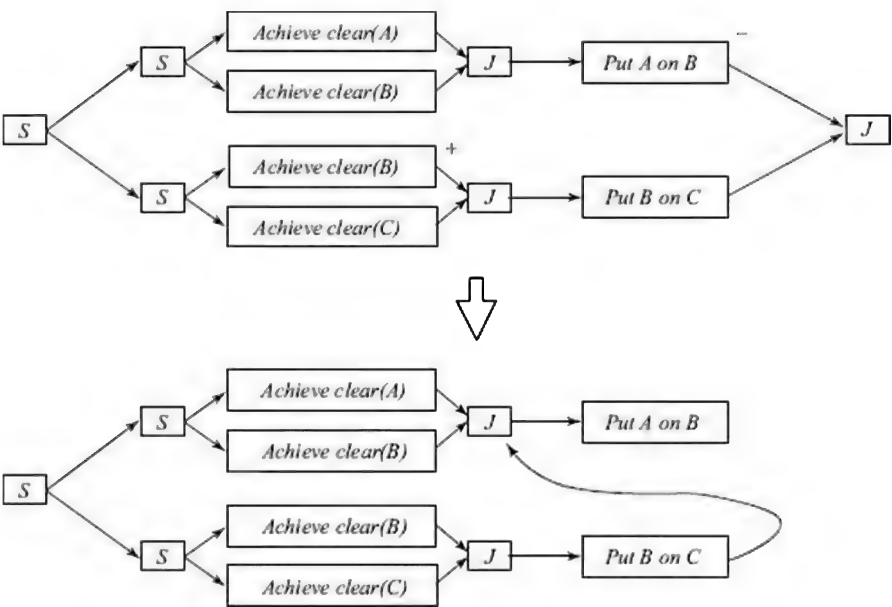


Figure 7.26 NOAH refines each of the two tasks and hands over control to the Critics. Critic: *Resolve-Conflicts* notices the conflict marked "+" and "-" on condition *clear(B)*. Suggests that the action labelled "+" be promoted before action labelled "-" to resolve the conflict (threat).

Having expanded both the tasks, it calls in the critics. Critic: *Resolve-Conflicts* notices the conflict marked "+" and "-" on condition *clear(B)*. It suggests that the action labelled "+" be promoted before action labelled "-" to resolve the conflict (threat), as shown in Figure 7.26. At this point the Critic: *Eliminate-Redundant-Preconditions* observes that the condition marked "+" in Figure 7.27 is redundant, and suggests removal of one of the two instances of the task. The figure shows this transformation.

Once the critics have done their job, NOAH goes back to refinement. It sees that it can clear block A by removing block C from it and expands that node. Again, the critic *Resolve-Conflicts* notices the conflict (threat) marked "+" and "-" on condition *clear(C)*. It suggests that the action labelled "+" be promoted, and the network is now as shown in Figure 7.28.

Now the critic *Eliminate-Redundant-Preconditions* observes that *clear(C)* marked "+" is being achieved twice, and suggests that one of the redundant nodes be removed. The resulting network is displayed in Figure 7.29.

The reader would have noticed that by now, NOAH has computed the best ordering of its subtasks, and all that remains to be done is to expand the "Put" actions into the lower level domain actions.

7.5.3 Hierarchical Planning

In hierarchical planning, high level planning operators are refined into low level ones. The systems described above illustrate various features one

would like in an hierarchical planner. The planner should be able to reason about the problem with high-level operators and then drill down into the details. Like *ABSTRIPS*, it should be able to first focus of critical predicates and ignore the others. Like *NOAH*, it should be able to refine the high level task into networks of low level operators, with more detailed preconditions. In general, the networks should be partial orders containing only the necessary ordering information, to enable the planner to deploy the least commitment ordering strategy demonstrated by plan space planners and by *NOAH*. And like Korf's macro operators, the final plans must be composed by putting together primitive actions from the domain.

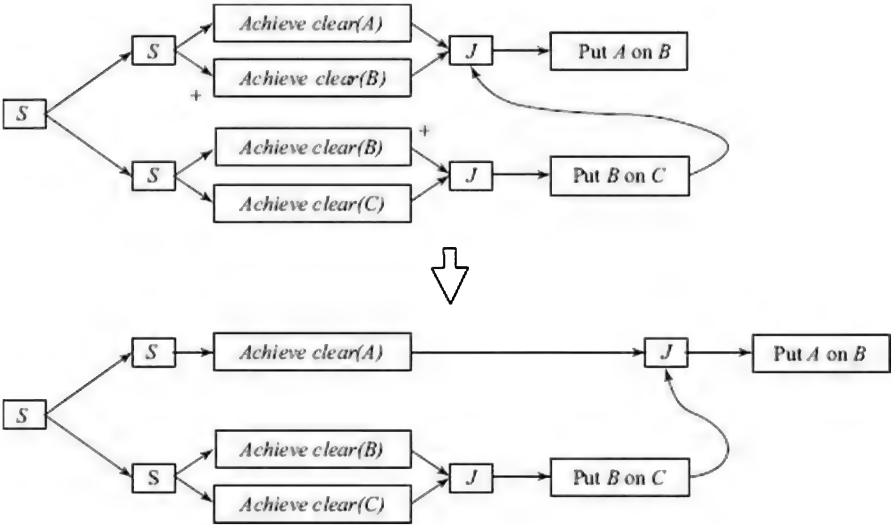


Figure 7.27 Critic: Eliminate-Redundant-Preconditions observes that condition marked "+" is redundant. Suggests removal of one task.

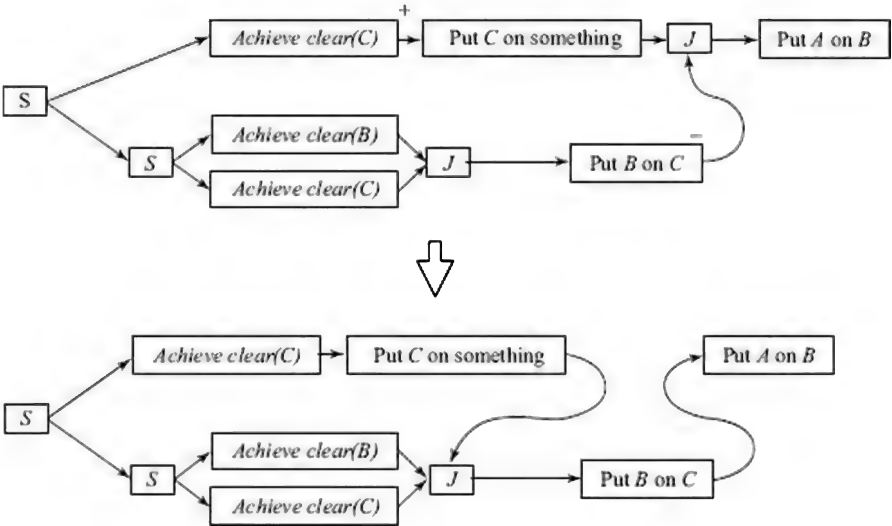


Figure 7.28 Achieve *clear(A)* is refined to moving *C* onto something after achieving *clear(C)*. Critic: Resolve-Conflicts notices the conflict marked "+" and "-" on condition *clear(C)*. Suggests that the action labelled + be promoted.

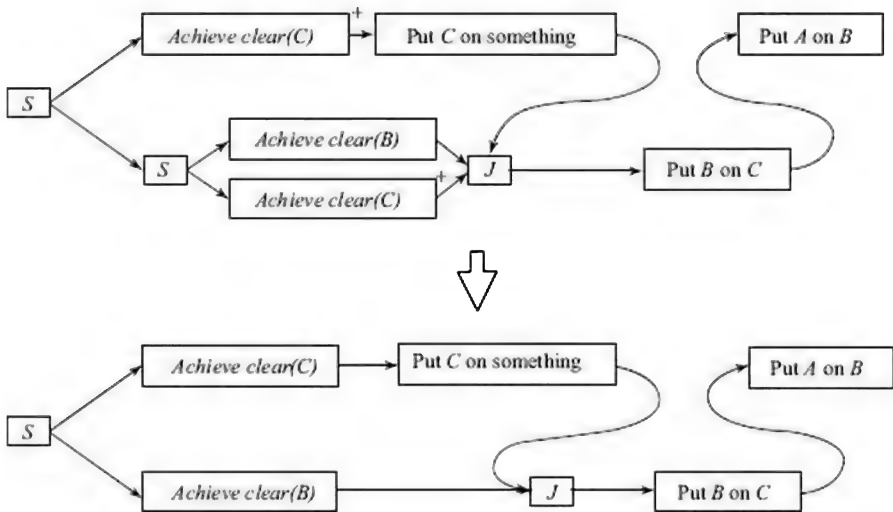


Figure 7.29 Critic: Eliminate-Redundant-Preconditions observes that condition marked + is redundant. Suggests removal of one task. The high level operator "Put *X* on *Y*" can be further decomposed into *STRIPS* actions. Blocks *B* and *C* need no actions to clear them. But by now, *NOAH* has found the correct order for the optimal plan.

The schematic diagram of the expansion of a high level operator is shown in Figure 7.30. The operator has its *own* set of preconditions and effects, shown with thick dashed arrows. The effects may be thought of as desired effects, because after refinement, the low level operators may also have other effects, shown with thin dashed arrows, which we can call 'side effects'. Like *ABSTRIPS*, searching and putting together a high level plan based on the preconditions and effects of the high level operator may be considered. But after expansion, other preconditions and effects may come into play. The preconditions may have to be satisfied, and the post-conditions may have to be inspected for threats and redundant conditions.

Similar approaches to hierarchical task network (*HTN*) planning have been reported in the literature (Wilkins, 1988; Erol, 1994; Tate, 1994). In a system called *SHOP2* (Simple Hierarchical Ordered Planner version 2) (Nau, 2003), the high level actions are called *tasks* and the decomposition of operators is done by *methods*. A *SHOP2* method constitutes of a name, the name of the task it decomposes, a set of subtasks that the method generates, and constraints between the subtasks. Planning involves a combination of method selection and task decomposition, until one is left only with primitive tasks and the task network generated is consistent.

The advantages of using hierarchical planning methods are that in principle, they allow the planner to start at a high level and work out the details selectively. Secondly, they make it possible for the users to feed in problem solving knowledge in the form of the high level operators, and thus exploit human generated knowledge, in addition to the learning of macros proposed by Korf. The last point, however, is a double edged feature,

because to exploit the features of *HTN* planning, one has to be able to devise appropriate high level operators, and algorithms to efficiently interleave the tasks of search, decomposition and constraint reconciliation, specially in domains where the lower level operators may have to be interleaved. But given the fact that planning by search at the level of domain level operators is hard, hierarchical planning is an approach where the combinatorial explosion can be contained. However, to do so, one has to be able to represent the problem and the operators at different levels of abstraction, and also to establish the relations between them.

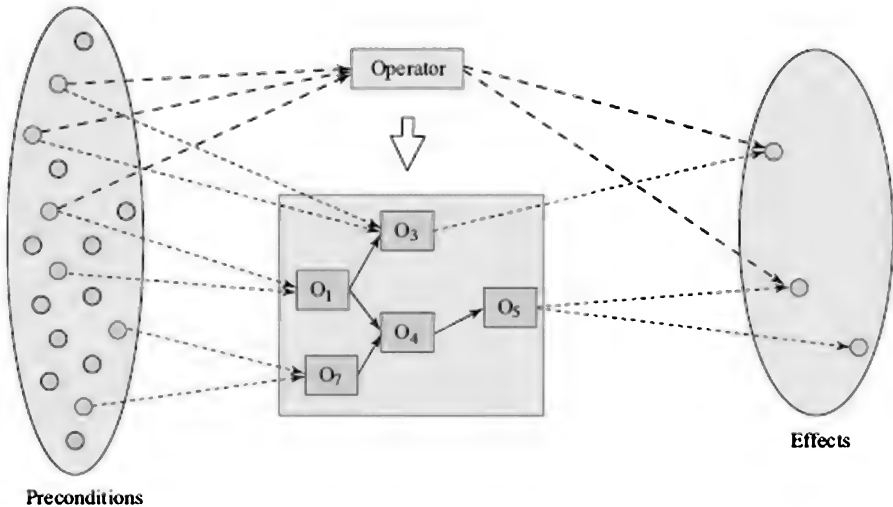


Figure 7.30 Schematic of a high level operator. The operator has its own preconditions and effects, and is refined to a partially ordered network of lower level operators.

This is essentially a problem of knowledge representation, and therefore *HTN* planners have a strong knowledge based flavour. Such knowledge can effectively compress the search space, as we saw in the Eight-puzzle example using macro operators. But in doing so, they may lose the generality and completeness of search. One of the challenges in the domain of planning is to combine the domain compression achievable through the deployment of knowledge with the flexibility and completeness of search. Ideally, a system should be able to exploit knowledge wherever it can, and fall back on search where it cannot. One can then think of complete planning systems that can become more and more efficient, as they have access to more and more knowledge. This knowledge could be generated through approaches to machine learning, but could also be acquired from external sources, like a teacher, or an experienced elder when the planner is in a social setup.

7.6 A Unified Framework for Planning

The planners described in this chapter represent themes that were explored up to the mid-nineties in the twentieth century. Most researchers tended to

view the different approaches as fundamentally different, until it was shown by Rao Kambhampati that all these approaches to planning could be viewed in a unified framework that he called *refinement planning* (Kambhampati, 1997). Significantly, he argued that the state space planning approaches too had a place in the unified framework in which the basic structure being operated upon is a partial plan.

The partial plan is a partially ordered network in which nodes represent operators, and edges represent ordering links and causal links. The ordering links are divided into two kinds. One, called *precedence* links, assert that one action happened before another. The other, called *contiguity* links, represent the fact that one action happened *immediately before* another. The two ends of the partial plan are the actions A_0 and A_∞ as in plan space planning described earlier. The *head step* is the last step at the end of a chain of contiguity links starting from A_0 , and the *tail step* is the first step in the sequence of contiguity links ending in A_∞ .

In addition, one can introduce a notion of state in two places that can be computed easily. One, called the *head state*, is the set of predicates obtained by progressing the “start state” till after the head step, and the other called the *tail state* is the set of predicates obtained by regressing the “goal set” across the contiguity links from A_∞ to the tail step. The partial plan representation is illustrated below in Figure 7.31.

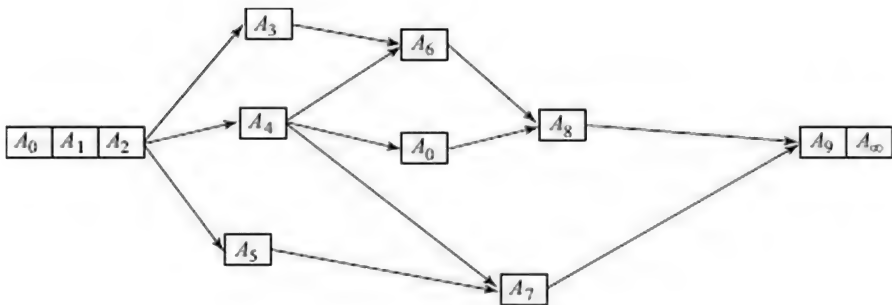


Figure 7.31 The ordering links in a partial plan. A_0 , A_1 and A_2 are linked by contiguity links, as are A_9 and A_∞ . The arrows show the precedence links. Action A_2 is the head step, and A_9 the tail step. The steps A_3 to A_8 are called the middle steps. The figure does not show causal links.

Thus, while the planner operates on a representation that does not include states, one can implicitly associate two states with the partial plan, and thus include the state space planning into the unified framework. Further, if one can include high level actions into the partial plan then we can have a unified framework in which all the different planning approaches can be incorporated. The framework described here has been adapted from (Kambhampati, 1997).

In the unified framework, we start with a partial plan like the plan in PSP containing the two actions (A_0 , A_∞). We can also think of this as a high level network of *NOAH*, made up of a single task that is to achieve the goal predicates. Thus, plans are like networks in *NOAH*, in which the high level tasks are made up of the open conditions in PSP. Refinement of this network can be done in any one of the following ways,

- Expanding a node like in *NOAH* or in *HTN* planning. A node is *replaced* by a network of lower level actions.
- Introducing new actions like in *PSP*. Addition of actions in *PSP* has a backward state space search flavour because it caters to open conditions of some nodes. One can also allow actions to be added by *MEA* like computing as in *ABSTRIPS*, which identifies some essential actions first, and worry about their placement later. From the *NOAH* perspective, this would mean refining a high level task.
- Imposing ordering constraints on some nodes of a partial plan as in *PSP* and *NOAH* in response to threats or conflicts.
- Adding actions with contiguity links to the head step or the tail step, as in state space planning.
- Imposing contiguity links on some actions added by *MEA* strategy, like in *STRIPS/GSP*.

Thus, in principle, a planner could interleave any of the refinement steps above.

We can also think of refinement as partitioning a set of candidate plans. Planning starts with the set of all possible plans. Remember, that this set is an infinite set even for finite domains, as long as looping is possible. A refinement step selects a subset from a set of plans. Let the refinement step choose a subset *A*, as shown by the dashed lines in Figure 7.32. We say that the refinement is *complete*, if the subset contains all the solution plans in the parent set. For example, the refinement step in the figure on the right is complete, because the selected subset contains all the solutions, shown as the shaded set. If the refinement operators we have are complete then the planner will find a solution directly through the refinement process. If the refinement step is incomplete then the planner may need to backtrack at some point, if a refinement contains no solution and try another subset.

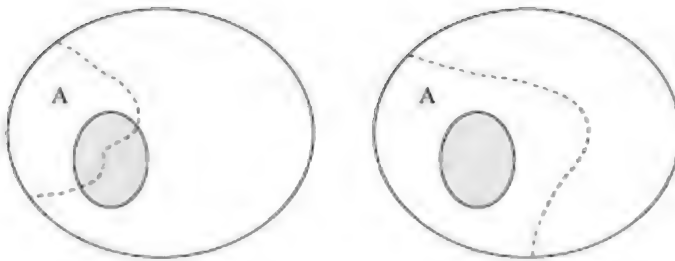


Figure 7.32 Refinement selects a subset of candidates. Subset *A* in the Venn diagram on the right contains the set of all solution plans, shown as a shaded subset.

We say that a refinement step is *progressive* if the selected subset is a proper subset. In other words, a progressive refinement step always prunes the set of candidate plans. We say that the refinement is *systematic*, if no candidate is visited more than once in the process of backtracking. This implies that the refinement step induces a partition and selects one of the components. As we have seen during the study of search algorithms, a systematic search implies that the algorithm will be able to terminate reporting failure when a solution plan does not exist.

Thus, one can think of the planner as successively selecting a subset of

candidates, till it has only solution plans. For the sake of completeness, the planner should be able to backtrack and try other selections. We can see that the more knowledge the planner has access to, the more dramatic the pruning in the selected set of candidates, and the lower the need to backtrack. In the example of solving the Eight-puzzle with macros, the need to backtrack was totally eliminated. In a complete knowledge based system, a solution is simply reconstructed or retrieved from memory, in this case the macro-operator table.

We shall look at the process of case based problem solving, which relies on storing and reusing experience, in more detail, later in Chapter 15. In practice, complete sets of solutions are rarely available, and case based systems have to contend with retrieving the best matching solutions available in the memory, and adapting them to the current problem. In some domains, it may be feasible to formulate heuristic rules to guide search, so that the chances of backtracking are minimized. The challenge, as a theme that we have been developing in this book, is to combine knowledge with search in a manner to solve problems as quickly and reliably as possible.

7.7 Discussion

In this chapter, we have looked at algorithms for planning in the simplest possible domains, known as the *STRIPS* domain. In this domain, the following assumptions hold.

The Actions are Instantaneous. Thus, we have only to worry about sequencing them, and there is really no notion of time. When actions are assigned *durations* then the total duration of executing the plan, the *makespan*, becomes a quality metric. For example, a recipe that uses a microwave for cooking is likely to produce your dinner faster. Further, when we talk of parallel actions then things become more complex, because of different durations. For example, while rice is being cooked on the stove, one could chop the vegetables, and knead the dough in parallel.

The Domain is Static. This means that the planning agent is the only agency of change. Thus while planning is being done, and while the plan is being executed, the world does not change.

The Actions are Deterministic. This means that the effects of actions when executed, are as intended. In such a world, the agent does not need to monitor the plan. In the real world, actions are not always deterministic, as anyone, who for example, attempted to walk on slippery ice would have discovered.

The Agent has Complete Information about the World. This means that the agent knows the state of the world completely. Often this is not the case in the real world. An agent planning a bid in an auction for example has no idea what the other agents may be planning to bid. Or an agent searching for a treasure does not know where the treasure is buried.

The Objective is Only on the Final State. Classical planning states the objectives as conditions on final state. In practice, there may be requirements on the plan trajectory too. For example, there may be a condition that if you open the refrigerator door, you must close it too.

The Goals are Categorical. A plan is valid if all the goal predicates are achieved. One may also want to introduce goals that are desirable, though not mandatory. This means, that one may be willing to relinquish some goal predicates, usually to save on some other cost metric. One may have, for example, a long shopping list, while going to the market, but may settle for a subset due to time or capacity constraints.

Each of the other assumptions may be relaxed, at the expense of increasing computationally complexity. We will look at some of the approaches to solve problems of some of the richer domains in later chapters. We will also look at some other approaches that take a completely different view of the planning problem, creating intermediate representations before addressing the planning task itself.

We have not used the notion of heuristic functions in this chapter, because the methods we are investigating are domain independent methods. However, given the fact that even the simplest planning domains are hard, there is an incentive to look for methods that will speed up the search. Towards this end, we will also look at the notion of domain independent heuristics in Chapter 10, which look at relaxed versions of planning problems to determine which of the choices to be made are more promising.



Exercises

1. Choose an interesting problem from the real world and pose it as a planning problem.
2. The Gripper domain of the International Planning Competition (IPC) (see <http://ipc.icaps-conference.org/>) is as follows. There are some number of named balls in a room. A two-armed robot, with two grippers, has the task of picking up the balls, taking them to the other room, and dropping them there. Define the Gripper planning domain in PDDL.
3. The Driver Log domain is as follows (see <http://planning.cis.strath.ac.uk/competition/domains.html>). Trucks area to be used to transport packages between different locations. The packages have to be loaded from the location where they are and unloaded at the destination location. The trucks have to be driven around by drivers, who may have to walk to the truck and board the truck before they can drive it. Express this planning domain in PDDL.
4. The Rovers domain of the IPC is a simplified version of the task that NASA's Mars rovers face. There are several rovers. Each rover is equipped with devices to gather data, and different rovers may have different devices. The rovers can move from waypoint to waypoint, gather data and transmit it back to the lander. Transmission may be

limited to waypoints that are directly visible. Paths may be of different types, which can be navigated by some different rovers. Formulate the above domain in PDDL.

5. Complete the BSSP algorithm described in the chapter to incorporate a check for looping by keeping a CLOSED list.
6. In Section 7.4, the given trace of the example in Goal Stack Planning the algorithm chose to solve for $on(B, C)$ first. What if the order had been reversed, and the algorithm had picked $on(A, B)$ first? Would GSP still have found a solution? Generate the trace to find the answer.
7. Given the initial state $\{on(P, T), onTable(O), onTable(T), AE, clear(P), clear(O)\}$, show how *goal stack planning* with STRIPS operators will achieve the goal $\{on(T, O), on(O, P)\}$. You may choose any order for the subgoals. What is the plan found?
8. Given the initial state $\{on(A, B), on(B, D), onTable(D), on(C, E), onTable(E), on(G, F), onTable(F), onTable(H), AE, clear(A), clear(C), clear(G), clear(H)\}$ show how *partial order planning* with STRIPS operators will achieve the goal $\{on(C, B), on(B, A)\}$. You may choose any order where a choice has to be made. What is the plan found?
9. Modify the blocks world domain so that one can stack two blocks on any given block. For example, it should be possible to say $\{on(A, D), on(B, D)\}$ as part of a state description. How will the preconditions for the *Stack* operator change?
10. Design a planning domain for playing the computer game Freecell or Solitaire available on many computers.
11. Enumerate the different kinds of *flaws* that a partial plan may have in plan space planning. Also, enumerate the ways of dealing with them.
12. Implement Goal Stack Planning and Plan Space Planning algorithms for the STRIPS domain. Test your algorithms for domains used in the International Planning Competitions (available on the internet).
13. Given the initial state $\{on(A, B), onTable(B), on(C, D), onTable(D), AE, clear(A), clear(C)\}$, show how *plan space planning* with STRIPS operators will achieve the goal $\{on(A, D), on(C, B)\}$. Draw the plan structure that the algorithm returns at termination.
14. The *Wumpus* world is a small artificial world with a strong legacy in literature¹⁵ that was suggested by Michael Genesereth as a testbed for intelligent systems, and used as an example in many games and the well known book by Russel and Norvig (2009). In one simple version, the task is for an agent to find as much gold as possible in a two-dimensional grid. Each time gold is collected, the agent's score goes up by some large amount (say 1000). The agent can move to neighbouring squares horizontally and vertically. This can be done by a combination of *move* and *turn*. In addition, the agent has a *grab* action for picking up gold. A shoot action can be done to *shoot* an arrow in some direction. Moving costs some points (say 2) as does shooting (say 50). The agent can also *perceive* some features that reveal what is in the neighbourhood. *Stench* indicates the presence of a Wumpus that would kill the agent if it entered the square. *Breeze* indicates that the agent is next to a pit in which it could fall and die. *Glitter* indicates the presence of gold in a neighbouring

square. *Bump* means the agent walked into a wall. *Scream* indicates that the Wumpus has been killed. None, one, or more of these percepts may be perceived in a square. Model the above game in PDDL.

15. Devise a set of actions for the cooking domain. Illustrate how a recipe for making an omelette can be expressed as a plan. Are there any actions that can be done in parallel?
16. Having to plan a trip, say from Chennai to Jaipur, the first thing one might do is to find suitable train and/or flight combinations between the two cities, and then fill in all the other actions. What kind of planning algorithms will allow one to do so?

-
- ¹ The problem description is the initial state description and the goal description. Each is a set of instantiated predicates defined in the domain description.
 - ² The name was probably an acronym for Stanford Research Institute Problem Solver, but has now itself entered the vocabulary of the planning community.
 - ³ Hanks and McDermott described it with a now well known and more dramatic Yale Shooting Problem, in which one has to determine whether a gun loaded earlier is still loaded when a shot is fired at a later time instance. (see for example http://en.wikipedia.org/wiki/Yale_shooting_problem).
 - ⁴ If one were allowed to have negative facts in the goal condition, for example (*not* (*on*(*B*, *C*))), then an additional condition would be needed stating that $g^- \cap effects^+(a) = \varnothing$.
 - ⁵ In PDDL3.0 one can even talk of conditions on the trajectory of the plan.
 - ⁶ The term 'linear planning' has also been used for planning systems in which a plan is a linear order on actions, and the partial plan is extended action by action in a linear order.
 - ⁷ Also known as an open goal, open subgoal, or an open condition. In the planning literature, the term 'goal' is often used for a precondition or subgoal.
 - ⁸ When the representation we are working with may contain variables, we will stick to the convention of identifying them by prefixing the question mark. These are different from the variables in algorithms which are programming variables, and are names for memory locations or values. These variables are more like variables in mathematics that stand for something unknown.
 - ⁹ Either $P = Q$ or both P and Q are of the form $R(?X_1, \dots, ?X_n)$ and the variables in P and Q can be unified. We look at unification in more detail in Chapter 12 on logic and inference.
 - ¹⁰ This could be computationally expensive though.
 - ¹¹ CHOOSE makes a choice non-deterministically. In practice, this will be a choice point where the algorithm may need to backtrack to.
 - ¹² In practice, we might in fact decide to "fly to Munich" and leave the details for the travel agent to work out, but we wish to illustrate one (problem

solving) agent solving the travel problem. Also, other considerations might warrant a train from Frankfurt to Munich.

¹³ Also known as the Towers of Brahma. According to Hindu mythology, Brahma gave the task of moving a 64-disk tower to pandits (priests) in Benaras (now Varanasi). The world is supposed to come to an end when they complete the task.

¹⁴ By level of abstraction, we mean the level in the tree generated by the hierarchy. The relation between parent and child may be different in different kinds of hierarchies.

¹⁵ See http://en.wikipedia.org/wiki/Hunt_the_Wumpus

Game Playing

Chapter 8

Playing games like *Chess* well has long been considered a hallmark of intelligence amongst humans. The game of *Chess* was most likely invented in India¹ (Murray, 1985), and has long been considered as a training ground for strategic thinking. *Chess* requires strategic and tactical skills, and in countries like the erstwhile USSR, it was actively promoted to help develop analytic skills (Kotov and Youdovich, 2002).

The computer science community quickly took up games from the earliest times. The first paper on computer chess was published by Claude Shannon (Shannon, 1950). In this paper, he discussed the merits and demerits of complete versus selective search. The first dramatic success in implementing games came, however, in checkers (also known as draughts). In the Dartmouth Conference (McCarthy et al., 1955) where the term AI was coined in 1956, one of the big exhibits was Arthur Samuel's checkers playing program. The striking feature of Samuel's *Checkers* program (Samuel, 1959) was that it learnt from experience, and grew better and better at the game, eventually defeating Samuel himself. In 1968, the British grandmaster David Levy had wagered a bet that no machine could beat him in *Chess*. The duration of the bet, fortunately for Levy, was 10 years. In 1989, Levy was soundly beaten by the computer program *Deep Thought* in an exhibition match. Computer programs improved steadily in performance, and by the mid-nineties were of world championship calibre. In 1996, IBM's *Deep Blue* (Campbell et al., 2002; Hsu, 2004) did achieve the feat of beating world champion Garry Kasparov, and in the following year beat him 3.5–2.5 in a six match series. But no one has yet bestowed the quality of intelligence² upon computers because of that!

Apart from the fascination that humans naturally have for games, they are very good as platforms for experimentation. Games provide a well defined environment in which states are intrinsically discrete. This means that one does not have to worry about processing input or effecting output in a complex environment, and can focus entirely on the decision making strategy. One can circumvent perception and action in the real world; problems that one would have to address if one were building a robot to play golf or tennis. Moreover, absolutely nothing is lost in abstraction. Furthermore, in games, the rules are well defined and success or failure can be measured easily. Good programs will beat inferior programs or humans in the game. Moreover, as we will see in this chapter, in spite of

the simplicity of the domain, they provide us with problems that are hard to solve.

Game playing is also interesting because it allows us to reason about multi-agent activity. The problem-solving activity studied in the preceding chapters is characterized by the fact that only a single agent is involved. Single agent situations do occur in the real world. For example, organizations can be seen as agents pursuing their goals in isolation. They could be in industrial organizations planning their design and manufacturing of products, or they could be government organizations planning infrastructure, or sending a man to the moon, or a rover to Mars. Or an individual may be planning a meal, or building a house.

Many real world situations, however, have multiple agents involved. In such situations, the problem-solving agents have to consider the actions of the other agents too, because they affect the world the agent is operating in. The agents may be collaborating with each other, they may be competing with each other, or they may be antagonistic to each other.

Interaction between agents has most commonly been studied by abstracting them as games. Games are formalisms that are used in various fields of study, ranging from economics to war. The common feature is that they attempt to devise models of rationality in the face of other agents being active. John von Neumann (Neumann and Morgenstern, 1944), the prolific computer scientist, is also credited with pioneering work in formalizing games and is often referred to as the father of Game Theory. The following example, known as the Prisoner's Dilemma, illustrates the kind of problems posed by von Neumann (Poundstone, 1993).

Prisoner's Dilemma

Imagine that the police have in their custody two suspects of a bank robbery, but no real evidence. Their only chance lies in a confession from one or both of them. They interrogate them in separate chambers, and offer each a lighter sentence if they confess, and betray the partner. Imagine, for a moment, that you are one of the two suspects. In the *normal* form of game (McCain, 2004) representation, a payoff matrix can be constructed, as follows:

Payoff: Yours / his	You confess	You deny
He confesses	-100 / -100	-200 / -10
He denies	-10 / -200	-50 / -50

FIGURE 8.1 A payoff matrix for Prisoner's Dilemma.

If both confess, they get -100 each. If only one confesses, he escapes with -10 but the other gets -200. If both deny, each gets -50 for possession of an illegal firearm. As one can see, the combined optimal

payoff is achieved by both denying and getting a total penalty of 100. However, there is the temptation to betray the other and escape with -10 . Seen from an individual perspective, the following *extensive* form³ of the game is represented as shown in Figure 8.2. Even though the choices are made concurrently, one views this as a sequential phenomenon in which the two players act one after the other. Since the outcome depends upon the other player's action, the rational choice is to choose a move in which the worst penalty is as small as possible. Thus, the safer choice for you is to confess because it could at worst lead to the penalty of -100 . If you deny the crime and the other player confesses, you will be in for a -200 penalty. Obviously, for habitual⁴ bank robbers, the more profitable strategy is to stand by each other and deny any involvement.

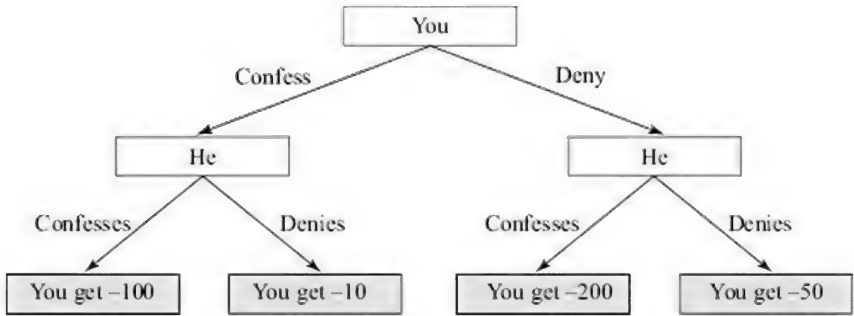


FIGURE 8.2 In the extensive form of Prisoner's Dilemma, the payoff is seen to be a function of the other person's choice after you have made your decision.

This can be perhaps seen more clearly by looking at an equivalent extensive form in which the other player plays first, as shown in Figure 8.3. If the opponent⁵ confesses, you are better off confessing yourself to get a penalty of -100 , instead of -200 . And if the opponent denies, you can get away with -10 by confessing.

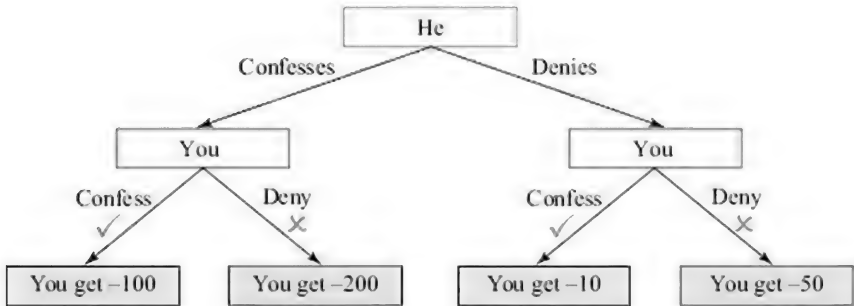


FIGURE 8.3 In an equivalent extensive representation, one can see that whatever the other player does; in each case, confessing is a better option for you.

Observe that in the *Prisoner's Dilemma*, the two players choose without really knowing what the other player has chosen. This is in

contrast to many situations where a player is aware of the opponent's choice while making a move. To distinguish between these two cases, one often draws the extensive form of the game (McCain, 2004) as shown in Figure 8.4. The idea is that the second player does not know which part of the tree he is operating in. This is depicted by merging the nodes at the second level into an *information set*.

One observation that one makes about the game is that simply acting rationally from one player's perspective, does not yield the best possible result. In this example, if both players were to deny then they would have both got a penalty of -50 , known as the Pareto optimal (after Vilfredo Pareto), but by *rationally* choosing to confess they both end up incurring a penalty of -100 each, reaching the Nash equilibrium (after John Nash). That is because rationality here is taking a pessimistic or conservative view trying to cater for the worst that can happen. In practice, human beings are often optimistic, they dream, they gamble and take chances, and they cooperate with each other. And on the average, they are better off as a result.

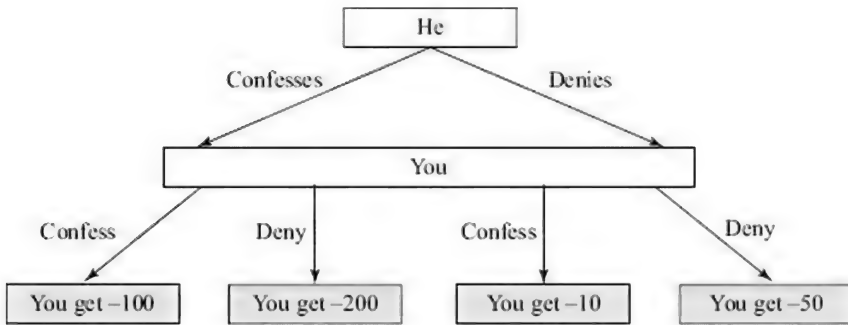


FIGURE 8.4 An information set is a node in an extensive form that hides the choice of the first player. The second player has to choose without knowledge of the move made by the other player.

Games⁶ thus are abstractions of interactions between agents. In the *Prisoner's Dilemma* example we saw above, the two players had to play simultaneously. Also, the strategy is concerned with choosing that one move. We will focus instead on board games like *Chess*, in which the two players play a *sequence of alternating moves* and the outcome is determined only when the game ends. Such games may be called *board games*. These games could in principle be represented and analysed by constructing the payoff table, but in practice, such a process would be cumbersome and computationally demanding. We will focus instead on more efficient strategies to play board games.

8.1 Board Games

The games we focus on primarily in this chapter are classified as

- two person,
- zero sum,
- complete information,
- alternate move, and
- deterministic games.

Two person games have exactly two players. In zero sum games, the total payoff is zero. One player's gain is the other player's loss. One wins, the other loses. In complete information games, both the players have access to all the information. That is, both can see the board, and thus know the options the other player has. In alternate move games, the players take turns to make their moves. In deterministic games, there is no element of chance in the moves that one can make.

All the above properties can be relaxed to produce more complex multi-agent environments. Adding a dice to a board game introduces an element of chance. The player cannot deterministically make a move. The essential feature of most card games is that they are multiple-player incomplete information games. Cards of other players are hidden, but the pack is known and finite. So a player does not know the options available to other players, but the possible options are bounded by knowledge of the cards that have not yet been played. In games like contract bridge, players draw inferences from the known information to glean as much additional information as possible. The game has four players, with partnerships of two each, and the strategy also involves communication of information between partners. Since this information is also available to opponents, the tactics often involve misinformation and deception too. Army generals fighting wars operate similarly with genuinely incomplete information. They also attempt to glean information about the resources and options of the enemy, and likewise the flow of information yields opportunities of misinformation and deception. War and spies provide a multitude of engaging stories concerning information exchange and deception⁷. Since both sides usually suffer casualties, wars can be seen as negative sum games, though sometimes one side may have some positive payoff. A price war, likewise, inflicts losses on the competing sellers, and can be modelled as a negative sum game, though if the buyers are included in the game too then it becomes a multi-player zero sum game. Cooperation is an example of a positive sum game, whether it is between students studying together for an examination, or when large corporations collude to jack up prices. Observe that some of the examples above do not have alternating moves, and neither is the outcome of their actions deterministic.

In the domain of recreation, *Checkers* (also known as *Draughts*), *Chess*, *Othello* and *Go* are examples of board games that have received the attention of programmers. One of the earliest to make a mark was Arthur Samuel's Checkers playing program (Samuel, 1959). It was a program that improved its performance with experience, and became news when it was able to beat its creator! The learning it did was essentially parametric *reinforcement learning*, tuning weights of its

evaluation function based on the outcome of each game. But it contributed to the wild notion of computers taking over the world, à la Victor Frankenstein's robotic creature (Shelley, 2001), in the novel written by Mary Shelley in 1818.

Amongst the incomplete information games, the most successful has been the implementation of *Scrabble*. In fact, the program *Maven*, now commercially available, can easily outplay human players (Sheppard, 2002). This is not surprising given that the machine can have access to a large vocabulary. In addition, it can speedily run through different combinations, and pick the one yielding maximum score. One could use a heuristic function that evaluates a board position by the points it yields, along with some measure of the cost of openings it creates for the opponent. A game that is met with less success is *Contract Bridge*. Though there have been several attempts at implementing the game (see: Throop, Frank, Khemani, Smith, Ginsberg, Sterling and Nygate), a truly world class player is yet to emerge.

8.1.1 Game Trees

We now look at some well known, search based algorithms to play complete information games. Our focus will be on alternate move games. In some games, players are allowed to move again in certain situations. For example, multiple captures in checkers, or potting a ball in snooker. We will assume that the move sequence can be modelled as a single (macro) move. We also assume the games to be two-player games. But the ideas presented can easily be extended to multi-player games. Finally, we assume that the game is a zero sum game. Our two players are traditionally named *MAX* and *MIN*, indicating that their goals are opposite of each other. The algorithms we study could be adapted to players whose goals are not diametrically opposite each other.

A game is represented by a *game tree*. A game tree is a layered tree in which at each alternating level, one or the other player makes the choices. The layers are called *MAX* layers and *MIN* layers as shown in Figure 8.5. Traditionally, we draw *MAX* nodes as square boxes and *MIN* nodes as circles in the tree⁸, and *MAX* is at the root. A game starts at the root with *MAX* playing first and ends at a leaf node.

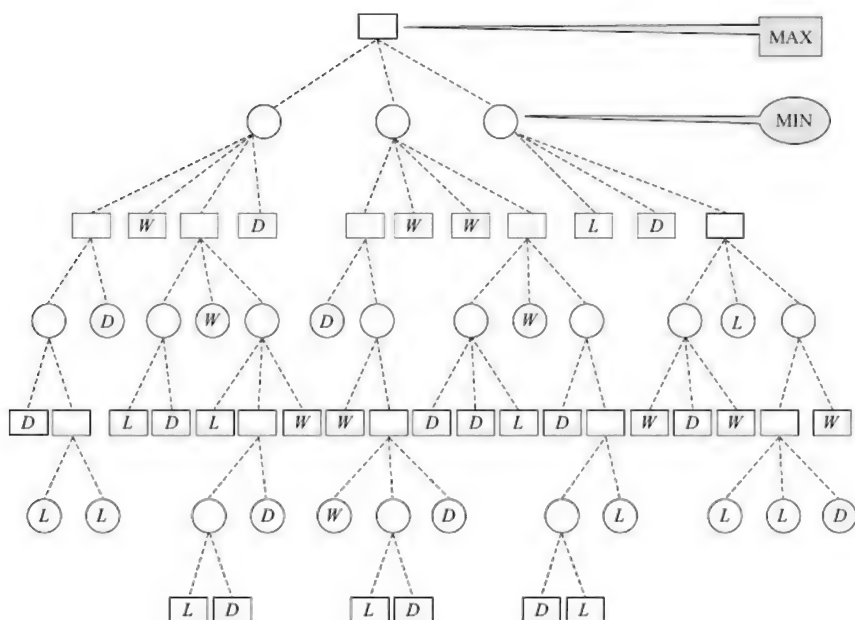


FIGURE 8.5 A small game tree. Leaf nodes are labelled with "W" for win, "D" for draw, and "L" for loss for Max. 1, 0, and -1 would be equivalent labels.

The leaves of the game tree are labelled with the outcome of the game and the game ends there. The task of each player is to choose the move when its turn comes. In the game tree, *MAX* chooses at *MAX* levels and *MIN* chooses at *MIN* levels. Thus, a game is a path from the root to some leaf node, chosen at alternating levels by the two players. For our zero sum game, the outcomes are defined by a set {win, draw, loss} and the values are as seen from the perspective of *MAX*, the player at the root. Thus, the value "win" means that *MAX* wins the game, and "loss" means that *MAX* loses or equivalently *MIN* wins. The leaves can also be labelled equivalently with numbers {1, 0, -1}. That is, it is a function that returns the outcome at a leaf node.

$value(leaf)$	= 1	if <i>MAX</i> wins
	= 0	if the game is a draw
	= -1	if <i>MIN</i> wins

One can now see the rationale of naming them *MAX* (the one who prefers the maximum valued outcome) and *MIN* (the one who prefers the minimum valued outcome).

Given a game tree, it is possible to analyse the game and determine the outcome when both players play perfectly. We can do this by backing up values from the leaf nodes up to the root. The backup rule is as follows.

Minimax rule

- If the node is a *MAX* node, back up the maximum of the values of its children.
$$\text{value}(\text{node}) = \max \{ \text{value}(c) \mid c \text{ is a child of node} \}$$
- If the node is a *MIN* node, back up the minimum of the values of its children.
$$\text{value}(\text{node}) = \min \{ \text{value}(c) \mid c \text{ is a child of node} \}$$

FIGURE 8.6 The *minimax* rule backs up values from the children of a node. For a *MAX* node, it backs up the maximum of the values of the children, and for a *MIN* node, the minimum.

The rationale of the rules is that given a set of choices with known outcomes, *MAX* will choose a move that yields the value = 1 (or *win*) if available, else 0 if available, and will have to choose a -1 (or *loss*), only if all its children are labelled with -1 . The backup rule for *MIN* is exactly the opposite, given that *MIN* is also trying to win the game. For *MIN* to win the game, the node must be labelled with -1 or *loss*. The *minimax* value of the game is the backed-up value of the root from all the leaves, and represents the outcome when both the players play perfectly. Figure 8.7 shows the above game tree with backed-up values. Observe that *MAX* wins *this* game when both players play their best.

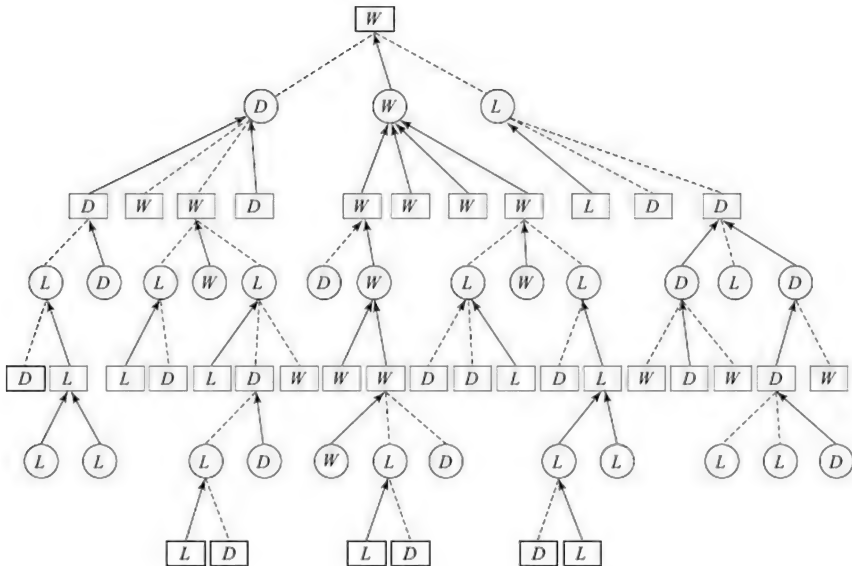


FIGURE 8.7 The game tree with backed-up values. Arrows show the values backed up. They identify the best move for each player. Where more than one move is best, all are marked. *MAX* wins the game because the backed-up value is *W*.

A game playing program is required to produce the moves for a player, traditionally *MAX*. Since the *minimax* value determines the best that *MAX* can do against a perfect opponent, this involves computing the *minimax* value, and the choice of moves that leads to it. The choices of a

player can be represented as a *strategy*. A strategy is a subtree of the game tree that freezes the choices for the player. A strategy can be constructed by the following procedure.

```

ConstructStrategy(MAX)
1  Traverse the tree starting at the root
2  if level is MAX
3      then Choose one branch below it
4  if level is MIN
5      then Choose all branches below it
6  return the subtree constructed
    
```

FIGURE 8.8 A Strategy for MAX.

The idea is that the strategy freezes the choices for the player, in our case MAX. That is, MAX has already decided the strategy, or the set of choices. The following figure shows two strategies for MAX in the given game tree.

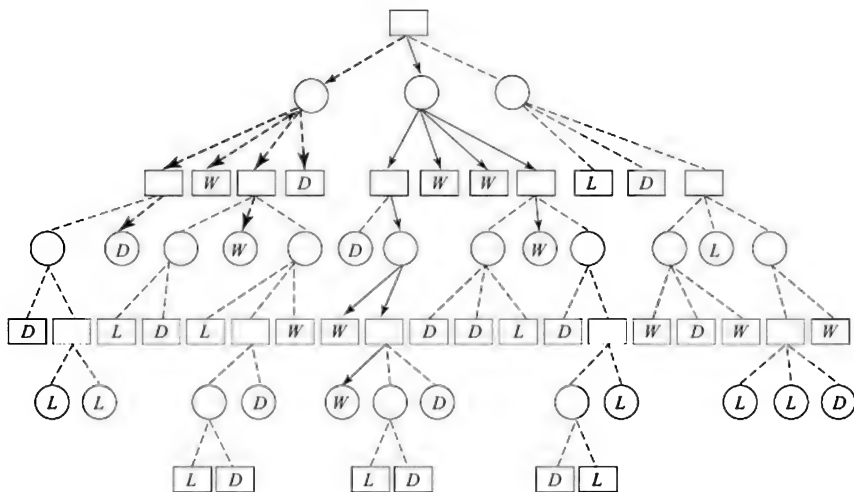


FIGURE 8.9 Strategies in a game tree are subtrees that represent choices of one player. The figure shows two strategies for MAX.

Once a player decides her strategy, the outcome of the game depends upon the opponent. Assuming that the opponent plays rationally, the value of the strategy for MAX will be the minimum value of a leaf node in the strategy, because that is where MIN will drive the game. Given a game tree, the *optimal strategy* for MAX is the strategy with the highest value. If this happens to be 1 (or win) then MAX has a *winning strategy*. In any case, the objective of both the players is to find their optimal strategies. Once both have chosen their strategies, the game played will be the path that is the intersection of the two strategies (the two

subtrees), as shown in Figure 8.10. The thick grey lines are a strategy for *MAX*, the thick dotted lines the strategy of *MIN*, and the thick black lines the resulting game path. Observe that the strategy for *MAX* shown in the figure is not an optimal one, leading to a draw.

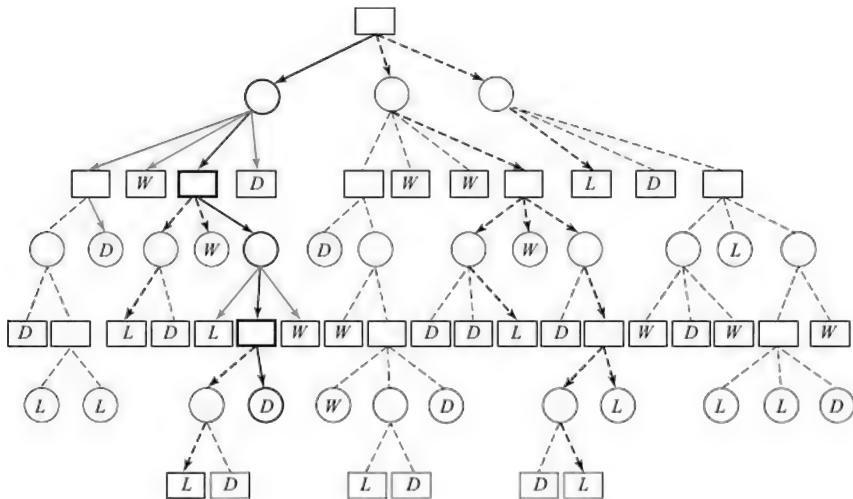


FIGURE 8.10 The subtree in grey arrows represents a strategy for *MAX*, and the one in dotted arrows, a strategy for *MIN*. The game played as a result of these two strategies is the path shown with black arrows, which is the intersection of the two subtrees.

Selection of the optimal strategy requires solving the game tree. Observe that the solution is a subtree, being a strategy, like the solution of an *And-Or* problem. In fact, finding the strategy is like solving the game tree as an *And-Or* problem. Each player has to choose her own moves, the *Or* choices, and cater to all possible responses by the opponent, the *And* nodes. With the cost of solving, the *And* node being the maximum of the costs of solving its children, instead of the sum, the question one might ask is can one use the *AO** algorithm to play games? One rarely hears of algorithms like *AO** being used to play games. The answer is that these algorithms could be adapted to play the games if it were possible to reach all the leaf nodes in the game tree. For most interesting games, the trees turn out to be too large to be traversed completely. Games with small trees can be completely solved. For example, the well-known game of *Tic-Tac-Toe* (also known as *Noughts and Crosses*, see Figure 8.11) is known to end in a draw when both players play correctly. But such statements cannot be made for games like *Checkers*, *Chess* and *Go*, because their game trees are too big. That is why these games are still fascinating for us to play. In *Chess*, for example, many people believe that *White*, the first to play, has an advantage. But this is only speculation. Let us see why.

The starting position in a *Chess* game⁹ has twenty possible moves for each player. As the game proceeds, the board opens up and the number

of choices increases further. Still further in the game as the number of pieces on the board reduce, the number of choices gradually comes down in the end game. It has been estimated that the average branching factor in *Chess* is thirty five, and that a typical game lasts about fifty moves. This means that the *Chess* tree has about 35^{50} leaves. This is roughly equal to 10^{120} leaves, a number that is difficult even to comprehend. One followed by one hundred twenty zeroes. Let us make a rough estimate of how long it will take to inspect just the leaves, forgetting about the internal tree. Let us assume that we have a fast machine on which ten billion leaves can be examined every second. Thus, it will take 10^{110} seconds to examine 10^{120} nodes. Like in Chapter 2, we can conservatively assume a hundred thousand seconds in a day, and a thousand days in a year. We will then need 10^{102} years, or 10^{100} (also known as a Googol) centuries. Compare this number with the total number of fundamental particles in the entire universe, which is estimated to be about 10^{75} , and one can see that the task of inspecting 10^{120} nodes is clearly in the realm of the impossible, with due respect to all who think otherwise¹⁰. Even if every one of these fundamental particles was a machine working in tandem with the others, it would still take 10^{27} years, which is much longer than the estimated age of the universe.

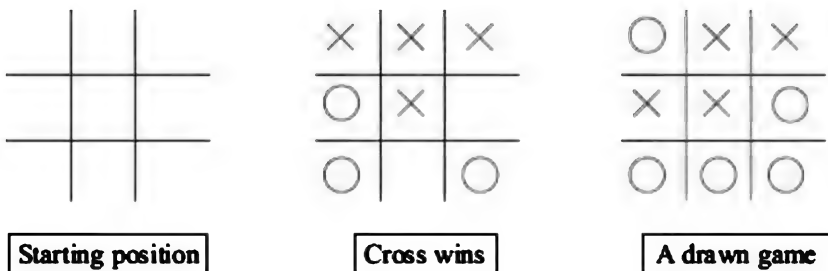


FIGURE 8.11 The game *Noughts and Crosses* is played on a 3×3 board. One player places a cross and the other a nought, alternately. The objective is to place three in a line first; row, column or diagonal. The first figure shows the empty board, the second a game one by Cross, and the third figure shows a drawn game.

The back of the envelope analysis done above, gives us a couple of insights. First, that even toy problems can be computationally hard. This means that real world problems will be harder to solve, unless they are posed very tightly. That is, only the minimal key features of the problem are identified and abstracted. That is why problem formulation is important to successful problem solving. The second insight is that we humans still do tackle many complex problems. And we do it with the aid of knowledge. Knowledge thus, has to be a key component for problem solving, if we are to build intelligent agents.

8.1.2 The Evaluation Function

Since we cannot inspect the complete game tree and compute the *minimax* value, we have to resort to other means of selecting the move to make. If we could have computed the *minimax* value, we would have selected the move that would have been *known* to be the best move. Now instead, we have to look for methods with which we will select moves that *appear* to be the best. This is done by using a function to evaluate the goodness of a state, like we did with heuristic functions in Chapter 3. In game playing terminology, we call it the *evaluation function*, and it tells us how good a given position (state) is from the perspective of *MAX*.

The outcome of a game is a value from the set $\{-1, 0, 1\}$. The evaluation function, however, is usually applied to an intermediate node, and we are not in a position to choose a value from the set, since the game has not ended, and we cannot evaluate the full tree. Instead, we define the range of the evaluation function as the real interval $[-1, 1]$. The extreme values -1 and 1 still represent wins for the two players. But other values *estimate* how close to winning each side is. For example, the value 0.5 says that *MAX* is better off than *MIN*, and 0.75 says that *MAX* is even more better off. A value of -0.9 says that *MIN* is doing very well. Note that the value zero says that both players are equally placed. It does not say that the game has ended in a draw. In practice, we extend the range to something like $[-10000, 10000]$ which is more conducive to devise evaluation functions. We will refer to it as $[-Large, +Large]$. Where do we get the values from? This is where the knowledge of an expert comes in. Generally, the evaluation function is computed as a sum of values of different features, and we add or subtract values for each good feature or bad feature. This kind of knowledge may be acquired from an expert, or one could devise experiments to learn from experience.

Typically, the evaluation function is split into different components. In *Chess*, for example, one may count the *material* value and add to that the *positional* value.

Chess players are used to assigning values to the pieces in the range of 1 to 10. Typically, pawns are valued at one point, knights and bishops about three points each, rooks about five points, and the queen about nine points. The *fighting value* of the king in the end game is equivalent to about four points. The material value of the king would be *large*, if it were counted, because its capture ends the game in a loss. *Chess* programmers however choose numbers in a larger range, thus enabling them to add positional values also more precisely. Thus, in the evaluation function of a program, a queen may be worth 900 points, a rook 500, a bishop 325, a knight 300 and a pawn 100. The material balance on a given board position is arrived at by adding the value of all the pieces on our side, and subtracting the value of pieces on the opponent's side. One may observe that the value of the evaluation function in the starting position is zero.

The positional part of the evaluation function looks at many aspects. These are concerned with mobility, board control, development, pawn

formations, piece combinations, king protection, etc. For example, two rooks in the same column have an added value; a pair of bishops is better than a bishop and a knight; knights are valuable in certain kinds of end positions. Traditionally, development and centre control have been given great importance; one's pieces must become as mobile as possible and either occupy the centre or control it. Pawn formations are also subject to evaluation; chained pawns that support each other are better than isolated pawns; pawns in the same column or opposing pawns head to head are not good; and as most players know, pawns heading for promotion have added value. The king needs protection in the opening and middle games, and structures like those obtained by castling are valued high; while in the end game, the king may be an offensive piece adding to the fighting strength. Pins, forks and discovered attacks also need to be considered while computing positional value. The evaluation function of the program *Deep Blue* has about 8000 components (Campbell et al., 2002).



FIGURE 8.12 The fork is a pattern in which a Knight attacks two pieces simultaneously. The opponent can only move one piece away. In the example, the Red Knight attacks the White Rook as well the White Queen.

The value determined by a component of the evaluation function could have also been determined by searching further. For example, if one ascribes a value to the existence of a fork *pattern* on a chessboard (see Figure 8.12), it says that a fork tilts the value in favour of the player by a certain amount. If this was not part of the evaluation function, the advantage would have become evident after a few plies search. In the example in the figure, the Red Knight attacks the White Queen and the White Rook simultaneously. White can move only one of the pieces away, and in the next move the Red Knight could capture the other piece, gaining material advantage, even if the knight is captured in turn. This estimate of material gain could be encoded as the value of the pattern. The key thing is that this value is available from the given board position directly, without having to look ahead. This also illustrates how knowledge can be used to trade off search.

One obvious way to play the game now would be to evaluate all the positions that result from the moves one can make, and choose the best one. This would be called *one-ply* look-ahead and is depicted in Figure 8.13.

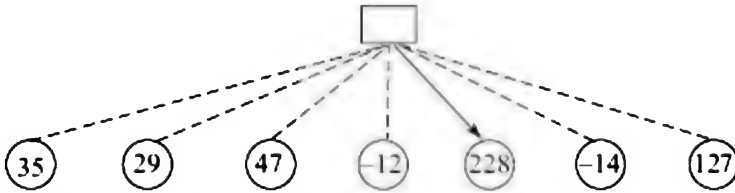


FIGURE 8.13 A one-ply look-ahead picks the best looking successor.

This would be fine if the evaluation function was very good. While it has emerged that grandmasters do store tens of thousands of *Chess schemata* and evaluate them directly (Sowa, 1983), it is difficult to devise an evaluation function that is good enough to play with one-ply look-ahead. In practice, *Chess* programmers rely on a combination of evaluation and look-ahead. While we cannot write programs to look ahead till the end of the game, we can still do so to look ahead a smaller distance. Figure 8.14 below gives you a feel of the exploding search space with a game tree involving four choices per board position.

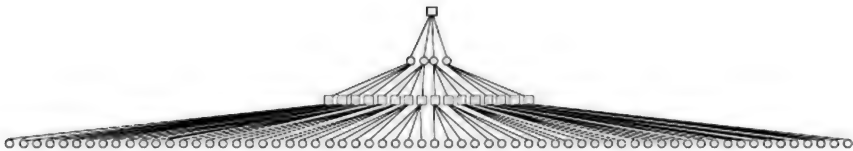


FIGURE 8.14 A game tree with a branching factor 4. A three-ply search needs to inspect 64 nodes. The next level will have 256, the one after that 1024, and the next one 4096; trees that are too large to draw on these pages. Most games have an even higher branching factor.

Look-ahead takes care of the combinatorial aspect of the game, like piece exchanges in *Chess*, which cannot be captured easily in a heuristic (evaluation) function, while the evaluation function provides a mechanism for evaluating the material and positional properties of nodes at the end of an incomplete look-ahead. The amount of look-ahead would basically depend upon the resources available to the program. The faster the machine, the more is the look-ahead possible in the same time. The more the program looks ahead, the better it is likely to play. Experts hypothesize that even with a simple evaluation function, a program that looks ahead fourteen to sixteen plies will play at a grandmaster level (Newborn, 2003). The extent of look-ahead is determined by the computing resources available. Most commercial programs do an eight- or nine-ply search. More sophisticated machines try and harness parallel computing with specialized hardware (Berliner, 1987; Campbell et al., 2002). One does not have to do a fixed look-ahead rigidly. We can write

programs to explore critical regions deeper. While playing competitive *Chess*, one is constrained by the clock. One can then do a flexible amount of look-ahead, depending on the amount of time available. This is typically done using an iterative deepening approach, like the one we studied in Chapter 2. The search component keeps searching deeper and deeper, and when the controlling program needs to play a move, it simply picks the best known move available.

8.2 Game Playing Algorithms

The basic procedure for game playing is then the following. Explore the tree up to a finite ply depth. Compute the evaluation function of the nodes on the frontier. Use the *minimax* backup rule described earlier to determine the value of the partial game tree, and the best move. Make the chosen move, wait for the opponent's move, and then again search for your best move.

```
GamePlay(MAX)
  1 while game not over
  2   do Call k-ply search
  3     Make move
  4     Get MIN's move
```

FIGURE 8.15 The top level game playing algorithm makes a call to search algorithm that backs up the evaluation function values from the horizon at k -ply depth. It makes the move that yields the *minimax* value, and after getting the opponents move, does another k -ply search.

If we could have searched the entire tree, the search would have to be done only once. But constrained to search only a part of a tree, we do a series of searches, one every time the program has to make a move. Every subsequent search starts two plies deeper than the previous one, and explores two more plies in the game tree. But, as shown in Figure 8.16, since it searches only below the chosen moves, it only looks at a fixed number of nodes at each level in the game tree. The complexity of algorithm can be depicted by the area of the search tree, which is proportional to the number of nodes in the tree. The figure below gives one an intuitive idea that the series of fixed ply searches explore only a small part of the entire game tree. Assuming that each search looks at P nodes, the game playing program will look at a total of $PN/2$ nodes during the entire time, where N is the number of moves made by both sides.

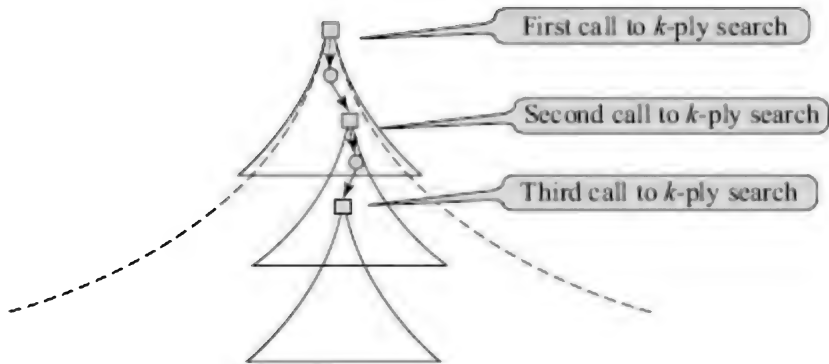


FIGURE 8.16 A game playing program does a k -ply look-ahead search for each move. It makes the best move, waits for the opponent to move, and does another k -ply search to decide upon the next move.

We now look at the basic algorithm for doing the fixed ply search. The algorithm uses an evaluation function $e(J)$ when considering the nodes at the frontier.

8.2.1 Algorithm *Minimax*

The algorithm *Minimax* searches the game tree till depth k in a depth-first manner from left to right. It applies the *minimax* rule (Figure 8.6) to determine the value of the root node. The following algorithm (Figure 8.17) is a recursive version adapted from (Pearl, 1984). The algorithm uses a test *Terminal(node)* to determine whether it is looking at a frontier node, and therefore should apply the evaluation function $e(J)$ instead of making a recursive call. A node is a terminal of a leaf node of the game, and will evaluate to one of $\{-Large, 0, +Large\}$ or it is a node on the horizon, and in that case the evaluation function $e(J)$ will be applied. Not shown in the algorithm is the implementation of the test for terminal node. It will need incorporation of a depth parameter k , perhaps passed along with the node, decremented at each recursive call. It will become zero when the node is on the horizon. This is left as an exercise (number 5) for the reader.

```

Minimax(j)
1  /* To return the minimax value  $V(j)$  of a node  $j$  */
2  if Terminal(j)
3  then return  $V(j) \leftarrow \epsilon(j)$ 
4  else for  $i \leftarrow 1$  to  $b$           /*  $b$  is the branching factor */
5      do
6          Generate  $j_i$  the  $i^{\text{th}}$  successor of  $j$ 
7           $V(j_i) \leftarrow \text{Minimax}(j_i)$  /* recursive call */
8          if  $i = 1$ 
9              then  $CV(j) \leftarrow V(j_i)$ 
10             else if  $j$  is MAX
11                 then  $CV(j) \leftarrow \text{Max}(CV(j), V(j_i))$ 
12                 else  $CV(j) \leftarrow \text{Min}(CV(j), V(j_i))$ 
13 return  $V(j) \leftarrow CV(j)$ 

```

FIGURE 8.17 The *MINIMAX* algorithm recursively calls itself till it reaches a terminal node. A terminal node is either a leaf of the game tree or a node at depth k . The algorithm does a k -ply search from left to right. Note that the recursive calls are of decreasing ply depth. One will need to keep track of depth of a node.

In the above algorithm, the *minimax* value is returned but not the best move that leads to that value. Since the objective is to play the game, the following version returns the best move. It calls the above *Minimax* algorithm for each successor of the root, and keeps track of the best move as well as the best board value.

```

BestMove(j)
1  /* To return the best successor  $b$  of a node  $j$  */
2   $b \leftarrow \text{NIL}$ 
3   $\text{value} \leftarrow -\text{LARGE}$ 
4  for  $i \leftarrow 1$  to  $b$ 
5      do  $V(j_i) \leftarrow \text{Minimax}(j_i)$ 
6      if  $V(j_i) > \text{value}$ 
7          then  $\text{value} \leftarrow V(j_i)$ 
8               $b \leftarrow j_i$ 
9  return  $b$ 

```

FIGURE 8.18 The algorithm *BestMove* accepts a board position and returns the best move for MAX. It calls algorithm *Minimax* with each of its successors and keeps track of which successor yields the best value.

Figure 8.19 depicts the tree searched by the algorithm *Minimax* and *BestMove* for a synthetic game tree. The tree is a binary tree, with two choices to each player at each level. The values for the evaluation at the 4-ply level have been arbitrarily chosen.

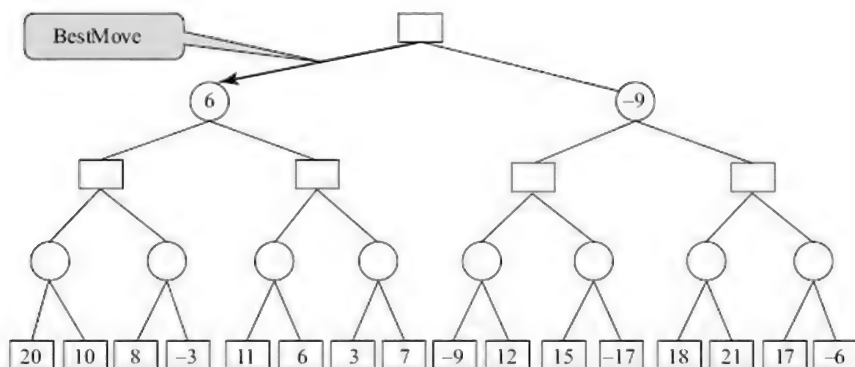


FIGURE 8.19 The algorithm *BestMove* calls algorithm *Minimax* for each of the successors of root, which computes the *minimax* value of each of them. It then chooses the best successor and returns that as the best move.

The *Minimax* algorithm above is the one that is doing the search. The *BestMove* algorithm is simply a modification to keep track of the best move found by *Minimax*. In Exercise 6, the reader is asked to modify the algorithm, so that the moves available to *MAX* at the next turn (after two plies) are stored along with their backed-up values. We will see in the following section how to exploit this information.

The algorithm *Minimax* finds the best move after searching the entire tree k -ply deep. There are, however, situations when it is not necessary to continue searching. This happens when it is known that searching further does have any scope of improvement. The simplest case is when a winning move has already been found, as shown in Figure 8.20.

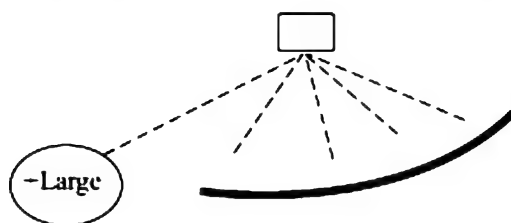


FIGURE 8.20 When a winning move has been found, the other options need not be explored.

8.2.2 Algorithm *AlphaBeta*

In the above figure, *MAX* has found a successor that evaluates to $+Large$. Since one cannot hope to improve upon, it does not make sense to search any further. We say that the search tree has been pruned. For pruning to happen, it is not necessary that a winning move has been found. It can also happen that during exploration it becomes clear that a particular child of a node cannot offer to improve upon the value delivered by a sibling. In that case, that node need not be explored further.

To understand this, we can view the game tree as a supply-chain process. At the top level, *MAX* has a set of *MIN* suppliers, from which it will select the one with the maximum value. Likewise, each *MIN* has *MAX* suppliers, from which the one with the lowest valued one will be selected. This process continues down the tree.

As the search in algorithm *Minimax* continues from the left to right, each node on the search frontier has been partially evaluated as shown in Figure 8.21. We will call the partially (or fully) known values of *MAX* nodes as *a* values. These values are lower bounds on the value of the *MAX* node. This is because that the *MAX* node will only accept higher values from the unevaluated successors. *MAX* nodes are also known as *Alpha* nodes. Likewise, *MIN* nodes are also called *Beta* nodes and store *b* values, which are upper bounds on values of the concerned *MIN* nodes. Remember that the *a* and *b* values are values that are already available to the respective nodes. They are not going to be interested in any successors that offer something inferior. Not only that, they are not going to be interested in any descendant that does not offer a better value.

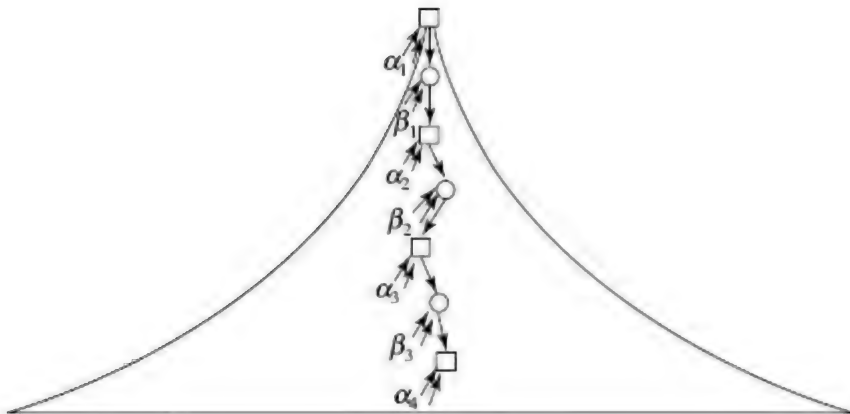


FIGURE 8.21 The search frontier contains a partial path in the game tree in which nodes have been partially evaluated. As this frontier sweeps to the right, these node will get fully evaluated.

The following example is from the *Noughts and Crosses* game. We assume a 2-ply search, in which the following evaluation function is used,

$$e(J) = \text{(numbers of rows, columns and diagonals available to MAX)} - \text{(number of rows, columns and diagonals available to MIN)}$$

We look at the progress of the search to understand the need to prune search, shown in Figure 8.22. The algorithm starts with *MIN* child *A* by placing a cross in a corner. The *MIN* node *A* looks at all children and evaluates to a value -1 . Note that while there are seven successors of *A*, only the five distinct ones are shown in the figure. Now the root has $a = -1$, which means that it will not go lower than -1 . In the figure, this is

expressed by the inequality $\alpha \geq -1$. It then turns to *MIN* child *B*. The first successor of *B* sends back a value -1 , which becomes an upper bound on the value of *B*. Since this is an upper bound and the root is already getting a value -1 , the root is not going to be influenced by *B*, and the *rest of the tree* below *B* can be pruned away. Note that even with *k*-ply search, where *k* is larger, the same pruning will happen as long as the backed-up values are as shown. It is also important to note that a node can be pruned only after it has been partially evaluated.

We call the pruning shown in the above figure an α cutoff. An α cutoff occurs below a β node, when it is constrained to evaluate to a lower (or equal) value than the α value above it. Correspondingly, when an α node has a lower bound (α value) that is higher than the β value of an ancestor than the rest of the tree below, it is pruned. This is known as a β cutoff.

In fact, this conflict between the bounds of two nodes need not be between a parent and child only. As described in (Pearl 1984), the *Alpha* node *J* in Figure 8.21 (with value α_4) will *influence* the root, *only* if it is greater than all the α values in the (*Alpha*) ancestors, and smaller than all the β values in the (*Beta*) ancestors. That is, the value $V(J)$ of a node *J* must satisfy,

$$\alpha < V(J) < \beta$$

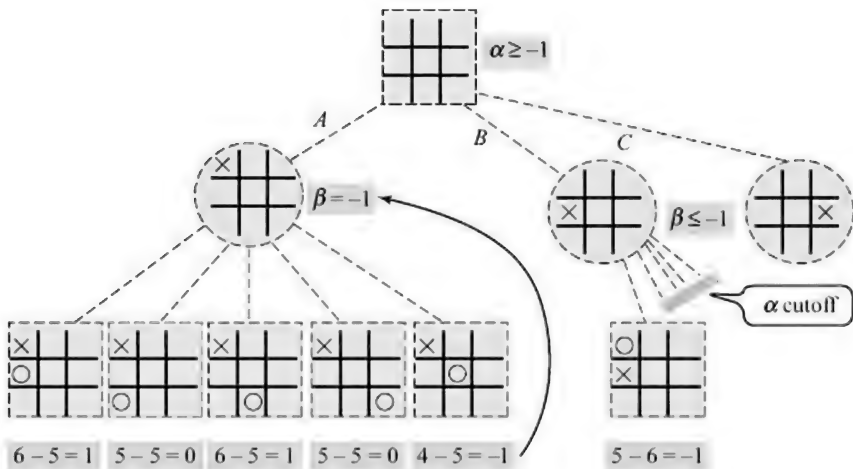


FIGURE 8.22 After evaluating the child *A*, the root node gets an α value of -1 . When it starts on move *B*, it sees a β value of -1 . Since this is an upper bound, the node *B* does not need to be explored further, and an α cutoff takes place.

where,

$$\alpha = \max \{\alpha_1, \alpha_2, \alpha_3, \dots\}, \text{ and}$$

$$\beta = \min \{\beta_1, \beta_2, \beta_3, \dots\}$$

When we are about to solve for a node *J*, we can propagate these bounds to the algorithm from its ancestors, and terminate the search if

these bounds are crossed. Note that an *Alpha* node can only increase in value, and can only cross a β bound. If it does then the search below the *Alpha* node is discontinued, and a β cutoff takes place. Likewise, a *Beta* node can only cross a lower bound α , and can be pruned using an α cutoff. The two kinds of cutoffs are illustrated in Figure 8.23.

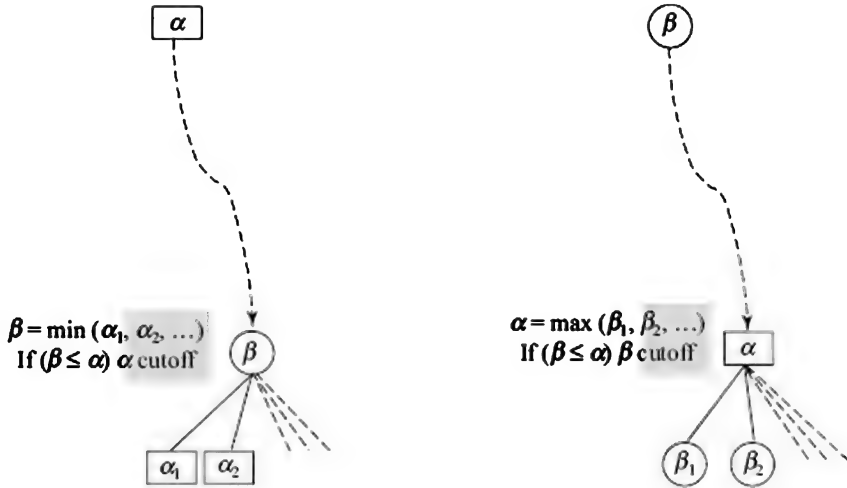


FIGURE 8.23 The α cutoff is induced by an α -bound from a MAX ancestor below a β -node. The β cutoff is induced by a β -bound from a MIN ancestor below an α -node.

The resulting algorithm known as the *AlphaBeta* algorithm is given in Figure 8.24 below.

```

AlphaBeta(j,  $\alpha$ ,  $\beta$ )
1  /* To return the minimax value of a node j */
2  /* Initially  $\alpha = -\text{LARGE}$ , and  $\beta = +\text{LARGE}$  */
3  if Terminal(j)
4  then return  $v(j)$ 
5  else if j is a MAX node
6      then for i  $\leftarrow$  1 to b      /*  $j_i$  is the  $j^{\text{th}}$  child of j */
7          do  $\alpha \leftarrow \text{Max}(\alpha, \text{AlphaBeta}(j_i, \alpha, \beta))$ 
8              if  $\alpha \geq \beta$  then return  $\beta$ 
9              if i = b then return  $\alpha$ 
10     else /*j is MIN */
11         for i  $\leftarrow$  1 to b
12             do  $\beta \leftarrow \text{Min}(\beta, \text{AlphaBeta}(j_i, \alpha, \beta))$ 
13                 if  $\alpha \geq \beta$  then return  $\alpha$ 
14                 if i = b then return  $\beta$ 

```

FIGURE 8.24 The *AlphaBeta* algorithm searches the tree like *Minimax* from left to right. It passes two bounds α and β to each call, and continues searching, only if the node has a value within those bounds.

The *AlphaBeta* algorithm is called initially with bounds $\alpha = -\text{Large}$, and $\beta = +\text{Large}$. As search progresses, these bounds come closer, and eventually converge on the *minimax* value of the tree. At any point, when it is recursively called with bounds α and β , it returns $V(j)$ if the value lies

between the two bounds. Otherwise, it returns a value β if J is an *Alpha* node, and returns α , if J is a *Beta* node.

An Example

The following figure shows the subtree in a 4-ply binary game. As in the earlier example, the values of the evaluation function have been filled in randomly.

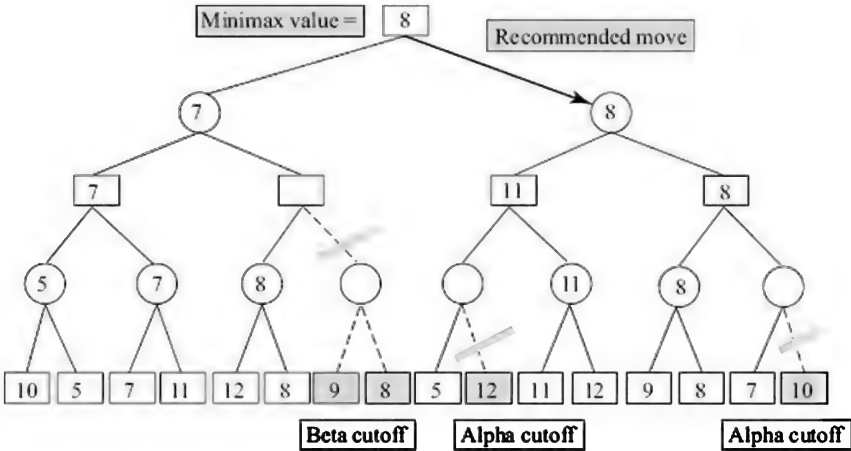


FIGURE 8.25 The *AlfBeta* algorithm evaluates twelve out of the sixteen leaves. It does one *b* cutoff and two *a* cutoffs as shown. The *minimax* value is 8.

The reader is encouraged to verify that the *minimax* value of the game tree is indeed 8, as reported by the *AlphaBeta* algorithm. This is irrespective of the values that are in the pruned leaves, shown as shaded nodes. Based on the leaves the algorithm has seen earlier in the search, which is in the left part of a tree, some leaves have no influence on the *minimax* value. The reader is encouraged to try different values in the shaded nodes and verify that the *minimax* value does not change.

Since the cutoffs are dictated by values of nodes seen earlier, the amount of pruning by *AlphaBeta* algorithm will depend upon the leaves seen earlier. If the *better* moves for both sides are explored earlier then the window of α -bound and β -bound will shrink faster and more cutoffs will take place. In particular, if the node whose value is backed up to the root, and which represents the best moves from both sides, is found earlier, the number of cutoffs will increase significantly. In the above figure, one can observe that the *minimax* value comes from the right half of the tree. Let us flip the tree about the root by reversing the order of the leaves and run the *AlphaBeta* procedure on the reversed tree. The resulting cutoffs are shown in Figure 8.26.

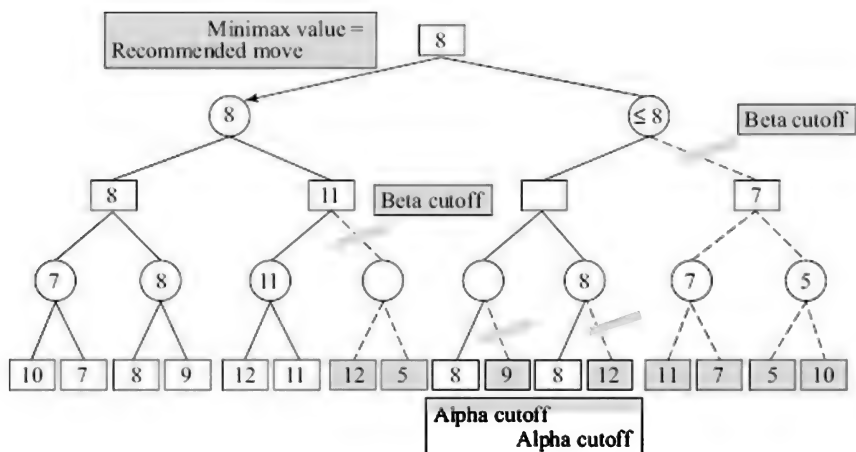


FIGURE 8.26 When the tree of Figure 8.25 is flipped about the root, the algorithm inspects only eight nodes out of sixteen.

As can be seen, the number of cutoffs in the same game tree, but with the order reversed, has gone up to eight, and only eight of the sixteen nodes are inspected. In Exercise 8, the reader is asked to construct a tree in which *AlphaBeta* is forced to evaluate all leaves, and then try the algorithm on the flipped tree.

It is evident that the performance of the *AlphaBeta* algorithm depends upon the order in which the moves are generated, and when the better moves are generated earlier, the cutoffs will be greater in number. But how do we generate the better moves first? One way could be to somehow put in domain-specific heuristics to order the moves. For example, in the *Noughts and Crosses* game, corner moves might be preferred over side moves.

Another, and a domain independent, way would be to use one instance of search to order the moves in the next instance of search. Let us say *MAX* is to play in some board position *X*. *MAX* calls the *AlphaBeta* algorithm and along with finding the best moves, also keeps track of the moves it explored at the third ply. These are moves it will start searching with the next time it has to make a move. *MAX* utilizes the *current* search to order the moves in preparation of the next one (see exercises 6 and 9).

Another way would be to give the algorithm a sense of direction, like in the transition from depth first search to heuristic search in Chapter 3. The algorithm *AlphaBeta* searches blindly from left to right. In the algorithm *SSS**, described in the next section, the search is guided towards the better looking nodes. The interesting thing is that the guidance does not come (directly) from a domain specific heuristic function. Instead, it is generated by a preliminary exploration of the game tree up to *k*-ply depth. This preliminary exploration yields some information which is used to guide the search towards better nodes. This notion of domain independent heuristics is an exciting one, and we shall

visit it again in the chapter on advanced planning (Chapter 10).

8.2.3 Algorithm SSS*

The main problem with the *AlphaBeta* algorithm is that it is sensitive to the order in which the moves are generated. This is because it always searches the game tree from the left to right, in an uninformed fashion. An algorithm which searches the game tree guided by heuristic information, is the SSS* algorithm developed by Stockman (1979).

The basic idea behind algorithm SSS* is the same high level notion of heuristic search that we have been pursuing in the earlier chapters.

Refine the best looking partial solution,
till the best looking solution is fully refined.

The difference is in the way in which a partial solution is evaluated. As described in Chapters 5 and 7, a partial solution stands for the set of complete solutions to which it can be extended. For the sake of completeness, it is imperative that search starts with and covers *all* potential candidate solutions till a solution is found. The job of the evaluation function is to help estimate the cost of a partial solution. Then, if the estimated cost of a partial solution is a lower bound on actual cost, the above strategy will terminate with the minimal cost solution, when a complete solution has the lowest cost. In the case of finding the *best* move for *MAX*, the optimal value is the *maximum* value. In this case, the estimated value of a partial solution should be an upper bound of the actual value of the partial solution.

The solution in the game tree search is a *strategy*. The task is to consider *all* strategies for *MAX* and pick the best strategy.

Consider the four-ply game tree that we have been using as an example. Remember that a strategy for *MAX* is constructed by choosing one move for *MAX* and all moves for *MIN*. The following figure shows one strategy in the game tree.

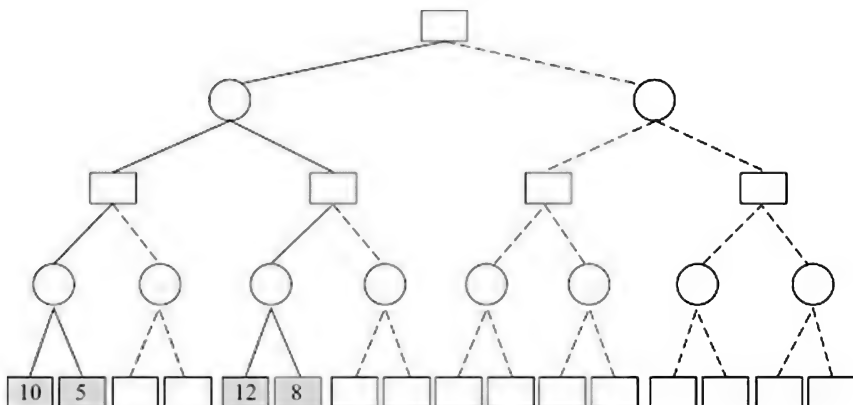


FIGURE 8.27 One of the eight strategies for *MAX* is the above 4-ply game tree. The value of this strategy is 5, the minimum value of the four leaves in this strategy.

The 4-ply binary game tree has eight different strategies. Two choices at the first level, and 2×2 subsequent choices at the next level (two for each choice of *MIN*). Both 5-ply and 6-ply have 128 (or 2^7) strategies and 7 and 8-ply have 2^{15} strategies. At the next level, each of the 2^{15} strategies can be extended in 2^{16} ways, yielding 2^{31} strategies.

The reader should verify that for a $(2n - 1)$ -ply or $2n$ -ply game tree of branching factor b , the number of strategies that *MAX* has to choose from is,

$$\text{number of strategies for } 2n\text{-ply look-ahead} = b^{\left(\frac{b^n - 1}{b - 1}\right)}$$

Fortunately, we do not have to enumerate and evaluate all these strategies. The algorithm *Minimax* looks at each node in the game tree exactly once, searching in a depth first manner. The algorithm *AlphaBeta* also searches in a depth first manner but prunes parts of the game tree. As we will see below, the algorithm *SSS** searches in a best first manner, and explores an even smaller part of the tree, visiting each node once.

The value of any given strategy for *MAX* is the minimum of the values of the leaves in that strategy. This is because the strategy freezes the choices for *MAX*, and we assume that *MIN* will play perfectly. Thus, if S is a strategy then,

$$V(S) = \min \{ V(L_i) \mid \text{where } L_i \text{ is a leaf in the strategy } S \}$$

This means that

$$V(S) \leq V(L_i) \text{ for any leaf } L_i \text{ in the strategy } S.$$

In other words, the value of a strategy for *MAX* will be equal to or less than the value of any given leaf in the strategy. That is, the value of a leaf node is an *upper bound* on the value of any strategy that it belongs to. For example, the leaf with value 10 in Figure 8.27 belongs to the two strategies, S_1 and S_2 depicted in Figure 8.28 below, and is an upper bound on both the strategies.

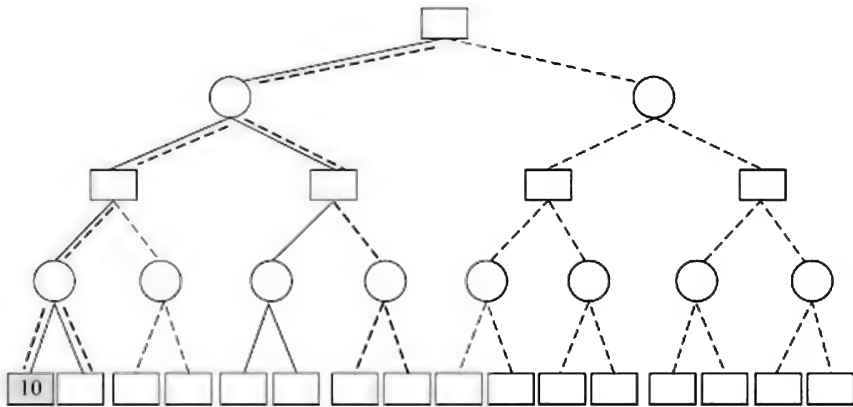


FIGURE 8.28 The node with value 10 belongs to two strategies shown in straight lines (S_1) and thick dotted lines (S_2).

Thus, the value of any given node is an upper bound on a set of strategies containing the node, and in fact any given node *defines* a *cluster* or set of strategies. We can also think of a node as representing a partial solution, or a partially developed strategy that can be extended in various ways. This represents a cluster of strategies and as more detail is added to the partial strategy, it becomes more refined, representing a smaller set, till eventually it becomes a single, fully refined strategy.

SSS*: An Example

If we can start the search with a set of nodes covering *all* the available strategies, and refine the one with the highest upper bound then when a fully refined strategy has a higher value than the other partial strategies, the algorithm can terminate. The procedure for finding nodes that will cover all strategies is like the one for constructing a strategy, only the choices are reversed. For *MAX* nodes, we select all choices, because we want to cover all strategies. For *MIN* nodes, we only choose one, since we are only constructing partial strategies that will be refined later. For the 4-ply binary game tree, four leaf nodes, each representing a cluster of two strategies, are selected as shown in Figure 8.29. The lone choice for each *MIN* node has uniformly been chosen as the leftmost one.

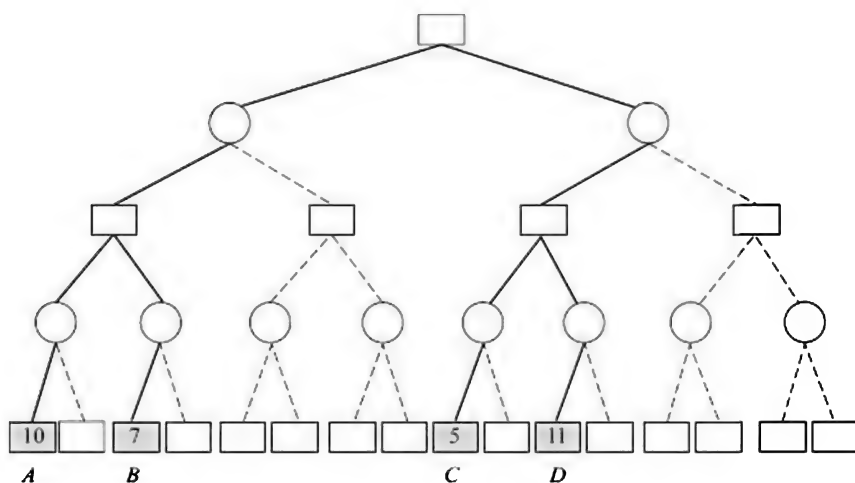


FIGURE 8.29 Algorithm SSS* begins by selecting a set of leaf nodes covering all strategies. All moves for MAX and one move for MIN is selected from the game tree. The four clusters are labelled *A*, *B*, *C* and *D*.

The four clusters labelled *A*, *B*, *C*, and *D* are shown by the four shaded representative nodes. The upper bounds on the four clusters are 10, 7, 5 and 11 respectively, the values of the four nodes. The best value possible for MAX is 11 in cluster *D*. Algorithm SSS* picks cluster *D* for refinement, and evaluates the next node in the cluster, as shown in Figure 8.30 below. Observe that in this example, SSS* has already headed off in a direction that seems to be best.

The new node evaluates to 12 as shown in the above figure. This does not change the upper bound of cluster *D* (remember 11 is also an upper bound). Cluster *D* is still the best and is again picked for refinement. The lowest MIN node in cluster *D* is fully evaluated, and the other refinement will have to come at the level of the next higher MIN node, by selecting the next child marked *X* in the above figure. Observe that the subtree below *X* is common to strategies *C* and *D*. Since *D* is better in the subtree where the two differ, *D* will always be better than *C*, and *C* will never be considered again. As shown in Figure 8.30 above, this is an α cutoff.

The subtree below *X* will only be of interest if it evaluates to a value less than 11, the upper bound on strategy *D*. Otherwise, the MIN node above it will not be influenced by its value. To refine the strategy *D* below node *X*, SSS* recursively calls itself with an upper bound of 11. It will continue evaluating that subtree, only as long as it is below this value. The recursive call generates clusters again for the subtree below node *X*. In our example, there are two clusters D_1 and D_2 as shown in Figure 8.31 below.

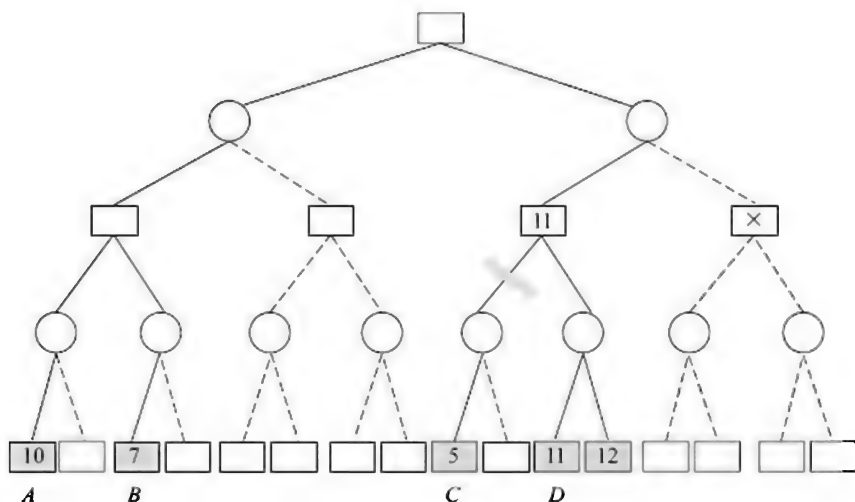


FIGURE 8.30 SSS* refines the best cluster *D* by evaluating another node that evaluates to 12. The upper bound on *D* remains 11. At this point, cluster *C* drops out of contention, and will never be looked at again. Cluster *D* is still the best and further refinement involves looking at another (the only one in this example) child of the parent Min node, marked *X* in the figure.

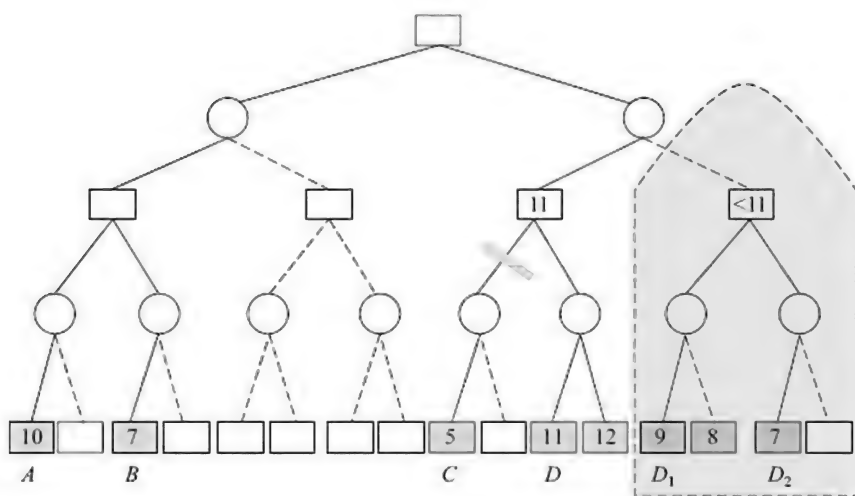


FIGURE 8.31 SSS* further refines the best cluster *D* by recursively calling itself with an upper bound of 11. The two (new) clusters are *D*₁ and *D*₂ with node values 9 and 7. It refines *D*₁ by inspecting another node in it, and *D*₁ becomes fully refined with a value of 8, which becomes the new upper bound on cluster *D*. Observe that *D* is now fully refined, but cluster *A* could still be better.

In the recursive call shown by the shaded envelope above, the clusters *D*₁ and *D*₂ begin with bounds of 9 and 7 respectively. The reader would have noticed that these clusters contain only one strategy each, and have resulted by splitting the cluster *D* that had two strategies. The better cluster *D*₁ is refined¹¹ by inspecting the remaining node in it, and

in the cluster is used as a representative of the cluster, in the following format.

Search node $p = J, s, h$ where

J is a node in the game tree

s is the status of p :

s is *SOLVED* if its *minimax* value is known, else it is *LIVE*.

h is the upper bound value on the cluster represented by node J

The algorithm, described in Figure 8.33, maintains a list *OPEN* of search nodes, sorted on the upper bound values, like our earlier algorithms for heuristic search in Chapter 3. That way the best node is always at the head of *OPEN*. But since the nature of the game tree is like an *And/Or* tree, it needs to treat each node differently based on whether it is a *MIN* node or a *MAX* node. The procedure is not recursive in nature, as the one described informally above. Instead, when a *MAX* node is *SOLVED*, and another sibling exists, the sibling is inserted in the *OPEN* list as a *LIVE* node along with the value from the *MAX* node. If no such sibling exists then its parent *MIN* can be labelled *SOLVED*.

Initially, the algorithm begins with the root node inserted with the label $\langle \text{root}, \text{LIVE}, +\text{LARGE} \rangle$. When the algorithm removes a nonterminal *LIVE* node from *OPEN*, it replaces it with *all* its children if the removed node was *MAX*, and *one* child if it was *MIN*. This *forward phase* terminates at the horizon. SSS* then applies the evaluation function $e(J)$ to the nodes on the horizon, labels them *SOLVED*, and arranges them in decreasing order of the h values (which in the first round are the values returned by the evaluation function $e(J)$). The node with the best h value is removed from *OPEN*, and its sibling added in Line 21 (for *MAX* nodes). Its sibling is added to *OPEN* with the lower of the two values. This is a refinement step in which one more leaf of the strategy is inspected. The reader should recall that each leaf is an upper bound on the value of a strategy. When it has finished with all the *MAX* siblings then the (*MIN*) parent is inserted in *OPEN* with the label *SOLVED*. If a *SOLVED MIN* node is picked by SSS*, its *MAX* parent can be labelled *SOLVED*, and its siblings removed. This is because its siblings have lower upper bounds (since the *MIN* node was at the head of *OPEN*). Contrariwise, when a *SOLVED MAX* node is picked by SSS*, its siblings are *added* to the *OPEN* with the same h value. This will be “recursively” solved by SSS*. The algorithm terminates when, like in AO*, the root is labelled *SOLVED*.

SSS*: The Example Revisited

Let us explore how the algorithm SSS* arrives at the solution shown in Figure 8.32, redrawn as Figure 8.34 with names assigned to the nodes explored by the algorithm. Initially, the algorithm begins with the root added to the open list.

$OPEN = (\langle root, LIVE, +LARGE \rangle)$

This is the only node in $OPEN$, and is a MAX node. It is removed from $OPEN$ and all (both) its successors added to $OPEN$. These are MIN nodes, and in turn replaced by one child. This process continues (Lines 7–13) till the horizon is reached. The open list now looks like this:

$OPEN = (\langle A1, LIVE +LARGE \rangle, \langle B1, LIVE +LARGE \rangle, \langle C1, LIVE +LARGE \rangle, \langle D1, LIVE +LARGE \rangle)$

The node at the head is removed and the evaluation function is applied to these four nodes one by one in Line 15. The new value of each node is the one returned by the evaluation function, and they are arranged in a sorted order as shown below.

$OPEN = (\langle D1, SOLVED, 11 \rangle, \langle A1, SOLVED, 10 \rangle, \langle B1, SOLVED, 7 \rangle, \langle C1, SOLVED, 5 \rangle)$

```

SSS*(root)
1 open ← ( < root, LIVE, +LARGE >)
2 repeat
3   Remove node p = < J, s, h > from head of open
4   if J = root AND s = SOLVED
5     then return h                      /* return when root is SOLVED */
6   else if s = LIVE
7     then if J is non-terminal
8       then if J is MAX
9         then for j ← b to 1
10            /* b : branching factor */
11            do /* J·j is the jth
12                child of J */
13                open ← Cons(< J·j, LIVE,
14                            h >, open)
15            else /* only first child
16                for MIN node */
17                open ← Cons(< J*1, LIVE, h >, open)
18            else /* J is terminal */
19            open ← Insert(< J, SOLVED, Min(h, e(J))
20                          >, open)
21        else /* status s = SOLVED */
22        if J = J0·j is MAX /* J is the jth child of J0 */
23        then if j = b
24          then open ← Cons(< J0, SOLVED, h >, open)
25          else open ← Cons(< J0*j+1, LIVE, h >,
26                          open)
27        else /* J = J0 * j is MIN */
28        open ← Cons(< J0, SOLVED, h >, open)
29        /* like an α cutoff */
30        Remove from open all successors of J0
31 until FALSE /* Repeat Indefinitely, exit when root is
32              SOLVED */

```

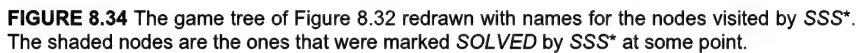
FIGURE 8.33 The algorithm SSS* maintains a sorted $open$ containing partial strategies and their upper bounds. It returns the $minimax$ value when the root is labelled $SOLVED$. The function $Insert$ inserts the triple after the last triple that has a higher value (as in the variation given in (Pearl, 1984)).

Now $D1$ is removed from $OPEN$, and since it has an unexplored

OPEN = (<D2, SOLVED, 11>, <A1, SOLVED, 10>, <B1, SOLVED, 7>, <C1, SOLVED, 5>)

OPEN = (<D12, SOLVED, 11>, <A1, SOLVED, 10>, <B1, SOLVED, 7>, <C1, SOLVED, 5>)

OPEN = (<D, SOLVED, 11>, <A1, SOLVED, 10>, <B1, SOLVED, 7>)



OPEN = ($\langle D_1, LIVE, 11 \rangle$, $\langle A1, SOLVED, 10 \rangle$, $\langle B1, SOLVED, 7 \rangle$)

OPEN = (<A1, SOLVED, 10>, <D₁1, SOLVED, 9>, <B1, SOLVED, 7>, <D₂1, SOLVED, 7>)

At this point, node *A1* becomes better and SSS* shifts attention to it. It

is replaced by its sibling A_2 with a value 5, the smaller of 10 and 5. $OPEN$ now looks like,

$$OPEN = (<D_11, SOLVED, 9>, <B_1, SOLVED, 7>, <D_21, SOLVED, 7>, <A_2, SOLVED, 5>)$$

Now D_11 is replaced by D_12 with a value 8, and that being the last sibling, the parent D_112 is labelled $SOLVED$.

$$OPEN = (<D_112, SOLVED, 8>, <B_1, SOLVED, 7>, <D_21, SOLVED, 7>, <A_2, SOLVED, 5>)$$

When the MIN nodes D_112 is removed, its parent D_1 is added as a $SOLVED$ node, and the MIN sibling D_21 removed from $OPEN$ (an Alpha cutoff).

$$OPEN = (<D_1, SOLVED, 8>, <B_1, SOLVED, 7>, <A_2, SOLVED, 5>)$$

Now since D_1 is the last sibling, its parent DD_1 is labelled as a $SOLVED$ node with the value 8. When this MIN node is removed, its parent root is added as a $SOLVED$ node with the value 8, and its siblings, represented by B_1 and A_2 are removed. Finally, since the root node has been labelled $SOLVED$, the algorithm terminates.

8.2.4 B^* Search

Another approach to selective search was developed by Hans Berliner, who had the distinction of being a world champion in correspondence chess¹², in addition to being a computer scientist. He developed the *Hitech* chess machine (Berliner, 1987) that used a specialized sixty four custom VLSI chip architecture. The B^* algorithm is a result of a quest for an algorithm that searches selectively in a *humanlike* manner. Berliner defines the necessary conditions for selective search as follows (Berliner, 1979),

1. It must be able to stop when a clearly best alternative exists at the root. This is done by comparison and is independent of the ultimate value of the best move.
2. It must be able to focus the search on the place where the greatest information can be gained toward terminating the search.

The first criterion is similar to the one in the AO^* algorithm (see Chapter 6). The second pertains to the heuristic aspect of search. Berliner provides an interesting perspective to the game tree search problem. He views the search space as a surface with geological features defined by an evaluation function in the (third) Z dimension. Only the game tree search space has layers and layers of ridges, as one looks farther ahead in the X and Y dimensions. Heuristic search is concerned with optimizing the value in the Z dimension (see Chapter 3). A strategic

outlook, on the other hand, is concerned with reasoning about choices in the X and Y dimensions. The algorithms discussed above *determine* the strategic outlook based on a predefined ply look-ahead search. In that sense, they are not selective, though the algorithm SSS* does refine partial solutions selectively. B^* search carries this idea one step forward, searching selectively from the very beginning. Further, it does not search a predetermined ply depth, but continues searching until the best option is *distinctly* better than the others.

To do this, it must have an idea of the goodness of a node. Remember that the algorithms described above apply the evaluation only at the leaf node on the search frontier. The values of the internal nodes are the backed-up *minimax* values of these leaves. The final decision is based on a subset of the nodes evaluated at the search horizon. Algorithm B^* also evaluates nodes using a small fixed ply look-ahead search, called a *probe* search. It uses the results of these evaluations to selectively explore certain move combinations, before deciding on the move to make. It is as if it is “*arguing and counter-arguing*” about some lines of play, using the probe searches to decide its move. Algorithm B^* assigns bounds on the values of each node as it searches forward. The node representation in B^* has the following elements (Berliner, 1979):

- The *RealVal*, which is the best estimate of the true value of the node.
- The *OptVal*, which is the optimistic value of the node for the side-on-move.
- The *PessVal*, which is the optimistic value for the side-not-on-move, backed up from its subtree, and
- The *OptPrb*, which is the probability that a certain target value can be achieved by future searches of the subtree rooted at this node.

The real values are determined by probe searches, using the *minimax* backup rule. Optimistic values are the best values¹³ of leaves of the same type (MAX or MIN). Pessimistic values are optimistic values for the opponent nodes. Optimistic values and pessimistic values are inherited by children from their parents during B^* search. The *OptPrb* values represent the probability that the *target value* can be achieved in that subtree. Probability distributions collect the information needed to make good decisions about what to explore and when to stop. The target value for a node, which has been determined empirically, is computed as,

$$\text{Target Value} = (\text{OptVal}(\text{2ndBest}) + \text{RealVal}(\text{Best}))/2$$

where the *Best* and the *2ndBest* refer to the children of a node with the best real value and the second-best optimistic value. The probability of a target value being reached from a given node is computed by linear interpolation between the real value, assumed to have probability 1, and the optimistic value, considered to have probability 0, of that node. If the former is 20 and the latter 250, then the probability of a target value of 100 is given by,

$$\text{Prob}(100) = (250 - 100)/(250 - 20) = 0.652$$

The probability of the best node is backed up for a *MAX* node, while for a *MIN* node, the product of the probabilities of the children is backed up. Figure 8.35 by Berliner shows a sample tree to illustrate these values. First, the root node is expanded to generate three children: *A*, *B* and *C*. The three are evaluated using the probe search, and their real values are 20, 18 and 10 respectively. The target value by the formula given above is $(40 + 20)/2 = 30$. Assuming that the probability of achieving real value is 1, the optimistic value 0, the probabilities of *A*, *B*, and *C* of achieving the target of 30 are interpolated as 0.875, 0.445, and 0.000 respectively. Note that the third one is set to zero.

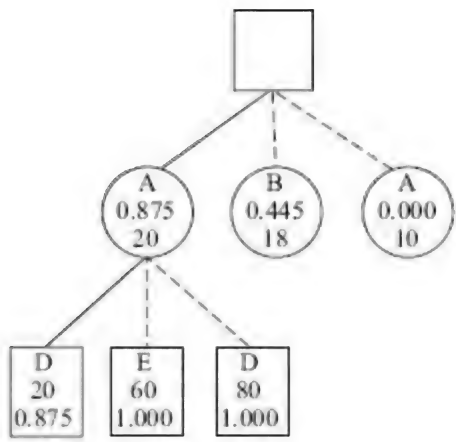


FIGURE 8.35 A sample tree generated by *B**. The nodes are arranged left to right in order of goodness for the respective player. The solid lines represent node chosen by *B** for exploration. Figure from (Berliner, 1979).

The following table shows the complete values for the nodes in the above tree. The most promising node '*A*' is expanded to generate nodes *E*, *F* and *G*. This is because *A* has the highest probability of achieving the target 30. The optimistic and pessimistic values are inherited from the parent, switching their labels. The real values are computed by a probe search. The probabilities of the three achieving the target 30 are 0.875, 1.000 and 1.000 respectively (since the latter two are "better" than the target). The product of these three, viz. 0.875, is backed up to the parent *A*, because node *A* is like an *AND* node.

Table 8.1 The values for the tree in Figure 8.34 (from (Berliner, 1997))

Node	Depth/Parent	OptVal	RealVal	PessVal	TargetVal	OptProb
A	0	100	20	Undef.	30	0.875
B	0	40	18	Undef.	30	0.445
C	0	25	10	Undef.	30	0.000
D	1/A	Undef.	20	100	30	0.875
E	1/A	Undef.	60	100	30	1.000
F	1/A	Undef.	80	100	30	1.000

The search proceeds in two phases. In the *SELECT* phase, the best descendent of a candidate is found based on the optimistic value, revising the target value on the way. Since the opponent will try and drive the value in the opposite direction, the phase *VERIFY* attempts to *refute* that the node is indeed a good one. In the *SELECT* phase, *MAX* acts as a *forcer* and the search attempts to find the most optimistic move that is likely to achieve the target value. *B** proceeds with the *SELECT* phase till the real value of a node becomes better than optimistic values of all its siblings that have the probability of achieving the target higher than a value known as *minAct*. Initially, *minAct* is chosen as 0.15 but it may be increased later in search to enable higher pruning. After the *SELECT* phase is over, let the real value of the second best sibling be a value *X*. The verify phase is called with a target value of $(X - 1)$, and it computes the probabilities of this target value. If the probability of the chosen node having this revised target is high then the *VERIFIER* has demonstrated that the chosen move is not the best, and a new *SELECT* phase will begin. In the *VERIFY* phase, *MIN* acts as the forcer and tries its best moves to show that *MAX*'s move is not the best one. If it succeeds, a new attempt by *MAX* will be embarked upon in another *SELECT* phase. If it fails, then search can terminate.

In the *SELECT* phase, the search proceeds from the root, selecting the best optimistic value at *MAX* level and the best real value at *MIN* levels. When it terminates, the *VERIFY* phase begins selecting nodes with the criteria of nodes reversed, starting at the chosen node. The two phases continue alternately, till a move has been found which has a high probability of being better than the other moves. The reader is encouraged to read the articles by Berliner for more details, for some of the chess-specific knowledge that has been built into the system, and the use of parallelism in the *Hitech* machine that used the *B** search algorithm.

We will conclude by observing the main differences between *SSS** and *B** search. *SSS** does a best first search over a fixed horizon. It starts by picking leaf nodes that represent all possible strategies, and refines the best one till the best strategy is fully refined. *B**, on the other hand, does selective search from the very beginning. In the first phase, it tries to optimistically find moves that look good from *MAX*'s perspective. In this phase, it may plunge down the tree till it feels that it has identified the best move from its choices. In the second phase, it pessimistically tries to check whether the move is really good. At all times, it keeps

optimistic and pessimistic bounds on the value of each node. When a move becomes distinctly better than the other options, it can terminate. This could happen after different amounts of exploration in different situations. It could happen quickly after a short search phase, or it may extend to longer periods when the surface is flat and not discriminatory. B^* can then progressively raise its *minAct* parameter to narrow down on its search space. The search space explored by B^* is schematically depicted in Figure 8.36 below, adapted from the paper by Berliner.

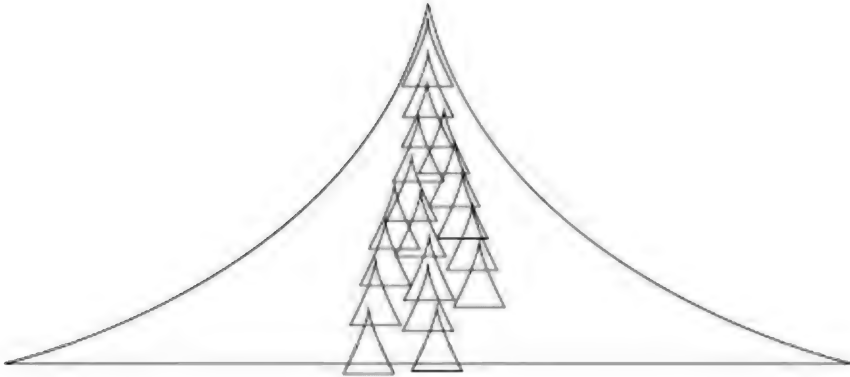


FIGURE 8.36 Schematic illustrating the search done by B^* , as compared to a full tree search by variants of the Minimax algorithm. The reasoning in B^* is done on the basis of a series of small probe searches as shown by the smaller envelopes. Figure adapted from (Berliner, 1979).

8.3 Limitations of Search

The game playing algorithms described above are constrained to search a limited part of a game tree. The constraints come from the limited time available to the program to find a move. Within this limited look-ahead, the search program has to rely on the evaluation function to decide the relative merits of the different moves. The evaluation function is basically some kind of perception function that looks at the board and returns a value that is indicative of how good the position is for *MAX*.

However, there are often situations where the evaluation function is unable to discriminate between different options because they are materially similar, and the consequences of the different choices are beyond the horizon of the search. This happens more often towards the end game, when the choices are typically fewer but more critical. There are some situations that are a kind of deadlock, where the interest of each player is in maintaining status quo. Plays in such situations explicitly try to delay meaningful activity, because changing the situation involves some kind of loss.

An example of such a situation in *Chess* is known as *Zugzwang* (Hooper and Whyld, 1992) in which the value of the board position critically depends upon who has to move. Figure 8.37 depicts two such

positions from (Flear, 2004).

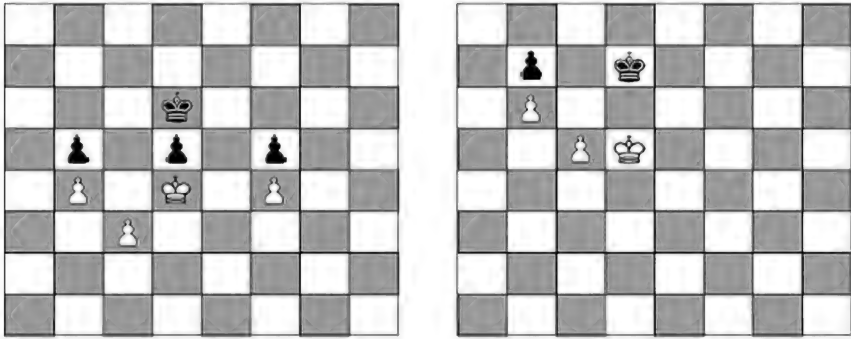


FIGURE 8.37 Two *Zugzwang* positions from (Flear, 2004). In the left position, if Black is to move in then the King has to withdraw protection from one of the three pawns, but if White has to move, Black is safe. In the position on the right as well, Black would rather not move. If White is to move however, the White King can be moved in a triangle (right, downLeft, up) to come back to the same position putting Black on move in *Zugzwang*.

In the first position on the left, the Black King is protecting the three pawns from being captured by the White King. If Black were to move then it would have to relinquish support for one of the pawns, clearing the way for White to win. If White were to move, on the other hand, the game would head towards a draw. Thus, the evaluation depends critically on who is to move next. In the second position on the right again, Black is in *Zugzwang*, and would rather not move, preferring it to be White's move. But in this position, even if White is to move, the White King can be moved in a triangle returning to the same position in *three* moves while Black is forced to oscillate and return to the same position in *two* moves. This manoeuvre, known as *triangulation*, has the effect of coming back to the same position, but now with Black to move in *Zugzwang*. Notice that it is difficult for an evaluation function to discriminate between these small but critical variations because neither the material nor positions change significantly. However, a more detailed knowledge based program could store such patterns to be acted upon in a specific manner.

A more critical illustration of the limitation of search and evaluation functions is the following contrived situation from (Seymour and Norwood, 1993) depicted in Figure 8.38 below. Human chess players quickly realize that the Rook on offer to White is poisoned, and refrain from capturing it. However, it is difficult for an evaluation function to resist the material gain of capturing the Black Rook because the negative effects of this action are far down beyond the horizon. The best computer chess playing program at that time, *Deep Thought II*¹⁴, grabbed the Rook when tested on the board position.

Human players evade the trap of capturing the poisoned Rook, because we do other things than look-ahead and perception. We create abstract representations of situations and reason over these representations. In the poisoned-rook test position, we realize that the

chessboard is partitioned into two regions, which because of the peculiar and unlikely pawn formations, and the absence of pieces like the Knight or the black square Bishop for Black that can break these formations, are separated by an impermeable boundary. All that White needs to do, given that White is materially much weaker, is to maintain this boundary by doing arbitrary moves of the King in its territory. Black offers a Rook to be captured by a White Pawn, a distinct material advantage, but that is the one move that White should not make. While White gains a Rook, the move disrupts the protective boundary and eventually loses the game. Once the pawn formation is broken, the remaining material strength of Black, the Bishop and the other Rook which were of no use, can now come forcefully into action.

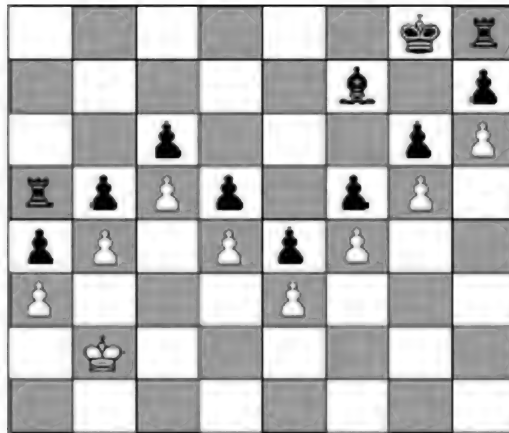


FIGURE 8.38 A 'poisoned rook'. In this unlikely chess position, human players quickly realize that it is (eventually) disastrous to capture the rook with the pawn. The program *Deep Thought II* apparently grabbed the rook given in this position (Seymour and Norwood, 1993).

8.3.1 Go

A game in which these limitations of search are brought to fore much more forcefully is the ancient Eastern game of Go, that originated in ancient China. The main problem with using search on Go is that the branching factor is much too high and the search tree is much too large. This is because the game is played on a 19×19 board called *goban*¹⁵, made up of the intersections of nineteen vertical lines with nineteen horizontal ones. The basic move in the game is to place a *stone* on one of the 381 intersections. The two players are traditionally Black and White, and they place a black stone and a white stone respectively. Black plays first on any one of the 381 intersections, and then White places a stone anywhere else on the board. Thus, we can see that the branching factor in the first thirty moves of the game is going to be higher than 350! The players play in turns, and a player has an option to pass, usually

used only in end games.

Go has been called a subtle game of territory (Iwamoto, 1977), and the objective is to have more stones on the board and *surrounding* more empty intersections (territory) than the opponent. The Japanese version of the game only counts the empty surrounded territory, while the Chinese version counts both the stones and territory. In addition to staking out territory, players can also capture opponent stones by surrounding them, and that counts as well in the final score.

Chess and *Go* are both board games, but while computer *Chess* has achieved spectacular success, computer *Go* is nowhere near any kind of human competence. One reason for this is that the search space is too large. The other is that the nature of evaluation is different. While pieces move around in *Chess*, they remain fixed on a *Go* board. Captures are few. *Go* players go through a process of learning in which they recognize more and more subtle patterns. They hardly employ look-ahead, except when it is a tactical necessity. Abstract knowledge is often expressed in the form of *Go proverbs* (Kensaku, 1966) and part of the learning process is to recognize when a proverb is applicable. In a comparison (Pinckard, 1992) of the three popular games in the east, *Chess*, the game of warriors and kings, is described as a game of “man vs. man”; the dice game of *Backgammon* (see the section on other games below), a game of “man vs. fate” contest; while *Go* the game of learning to recognize subtle patterns is billed as a “man vs. self” contest.

We now briefly look at some of the other games that are not two person complete information board games, like those that have been addressed in this chapter so far.

8.4 Other Games

The following games all differ from the games dealt with in the preceding sections in some way or the other. What is common between them is that for these games, it is not possible to construct a game tree to look ahead into the future. We discuss three interesting games: *Bridge*, *Scrabble* and *Backgammon*. Of these, the first two are incomplete information games, in the sense that the players do not know the holdings of the opponent, and the third has an element of chance in the game introduced via the throw of dice. Another distinguishing feature of *Bridge* is that it is not a two person, but a four person game, in which there are two teams of two players each. *Scrabble* can be a multiplayer game too.

8.4.1 Bridge

Contract bridge is a game played with a pack of fifty cards. The pack contains four suits: spades, diamonds, hearts and club. Each suit contains thirteen cards which form an ordered set {2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace}. The pack is dealt out amongst four players

traditionally called North, South, East and West. North and South are partners and so are East and West. The basic unit of play is a *trick*. A trick begins with a designated player playing a card, and the others follow in clockwise succession. They are required to *follow suit* or play a card in the suit led by the first player, unless they do not have one, in which case they can play any card. The highest card of the led suit played in that trick wins the trick, unless someone has played a *trump* card, in which case the highest trump card wins the trick. The corresponding player has to lead the card for the next trick.

The game is played in two stages. The first stage is an auction in which the players bid to make a certain number of tricks. A bid includes the number¹⁶ of tricks the side is bidding for as well as a suit to be the trump suit. If the final contract has a trump suit then all the cards of that suit are deemed to be higher than all other cards. Then if a player does not have cards in a led suit, she can *ruff* and win the trick with a trump card. A contract may also be bid without naming trumps, and is known as a *no trumps* contract.

The bids that a player can make are from an ordered list BIDS as follows:

$$\text{BIDS} = \langle 1C, 1D, 1H, 1S, 1N, 2C, 2D, \dots, 7H, 7S, 7N \rangle$$

where *C* stands for clubs, *D* for diamonds, *H* for hearts, *S* for spades, and *N* for no trumps. A player can only choose a higher bid than the previous one. A player may also choose to PASS anytime. In addition, the following bids may be used conditionally. If the previous nonPASS bid is by an opponent and from the set BIDS then a player may use a bid called DOUBLE. Finally, if the previous nonPASS bid is a DOUBLE by an opponent then a player may bid REDOUBLE.

The side that bids the highest gets the contract, after which one designated opponent has to start play by leading a card. The next player puts down her cards face up and is known as *dummy*. She does not play any more role in the proceedings and her partner, known as *declarer*, decides what cards the dummy will play. Thirteen tricks are played out after that, the winner of each trick starting the next trick. The objective of the declarer is to fulfil the contract and perhaps win more tricks, and objective of the other side, known as *defenders*, is to prevent her from doing so. The degree to which they succeed in their objectives determines their score on that deal. In tournament play, this score is compared with the other players on the same deal, and *that* decides the winners and the losers in a bridge event. It is not the side which Lady Luck favours with good cards, and consequently achievable good contracts.

The Auction

The first phase of a *Bridge* deal is the auction. The auction is a sequence

of bids by players in a clockwise order, according to the rules described above. It starts with the player who has dealt the cards and ends when three players pass in succession. The contract becomes the last bid from the set BIDS in the auction, with the declarer being the first player in that side who named that suit (or no trumps).

Ostensibly, the meaning of each bid is that the player is bidding for a certain contract. However, the auction in a typical game is not so straightforward. This is because, a player can only bid by looking at her cards, but is bidding for the side which includes her partner as well. It is the combined tricks won by the side that go towards fulfilling the contract. It would help tremendously if a player knew her partner's, and opponents' cards, but that is not the case. Since it is only the *last* bid from the set BIDS that finally determines the contract, the earlier bids need not be used to simply compete for a contract. This is specially so because there is a premium on bidding certain high contracts, known as *games* and *slams*, and players would like to bid them when the combined strength of the cards permits it. Therefore, most teams devise *bidding systems*, which define an alternate semantics of bids apart from the legal one binding for a contract. Thus, usually while on the surface, a bid is a bid for a particular contract, it usually carries some encoded information¹⁷ meant for the partner about her hand. And this is not something nefarious but within the rules of the game, and the bidding system has to be revealed to the opponent. A linguistic approach to model bidding has been explored in (Jamroga, 1999, 2000).

Bidding then is not merely making a bid for a contract. It involves other things as well. The primary task is to communicate information between partners to try and know enough to bid for the right contract. For this to be done, bidding systems encode information, typically about the number of cards in a suit or information about high cards, for the players to judge the right contract. But when communication is taking place, the opponents are there eavesdropping as well. They can also make use of the information being exchanged. This may motivate a process of deception, when a player intentionally conveys false information to mislead them. Bidding systems are often sophisticated enough to allow one player to ask the partner for some specific information. The players need to manage the resource of the bidding space available to exchange as much information as possible. The opponents may pitch in with nuisance bids, known as preemptive bids, to gobble some bidding space.

The process of bidding then is the process of encoding and deciphering information in bids, often in the face of resource constraints and interference from opponents. A player has to make inferences and *imagine* the most likely hands that a partner, and opponents, can have and arrive at an estimate of the best contract.

Finally, the success of bidding is not known after the auction is over. It is only reflected in the final score that the side gets after the play phase is over. For this reason, bidding is considered the hardest part of the game, and is usually a discriminating factor between novices and experts.

The main tasks in bidding are the following:

- Interpreting the meaning of bids of other people and constructing some representation of their card holdings. Note that this representation can only be an approximate one.
- Using the information inferred about the holding of other players, either making bids conveying more information, or asking for more information, or deciding the contract to be played.
- Making interference bids designed to try and disrupt opponents' communication. Note that one must be careful not to be saddled with a bid that has a high penalty for being unfulfilled during play.

Most known bidding implementations use rule based approaches. A few sample rules are illustrated below.

Rule: Feature rule Balanced-hand

IF

If no suit has less than two cards

If no suit has more than four cards

THEN

The shape of the hand is balanced

Rule: Bid rule open-1-N

IF

The shape of the hand is balanced

The hand count is between 15 and 17

No one has bid so far

It is the player's turn to bid

THEN

Make a 1 no trump bid

Rule: Bid rule open-1-Major

IF

It is the player's turn to bid

No one has bid so far

The hand count is between 12 and 21

Number of cards in major (spades or hearts) is five or more

Major is the longest suit

THEN

Make 1 spade or 1 heart (as the case may be) bid.

Declarer Play

During the play phase, there are three players involved. The declarer plays her cards as well as the dummy's cards. The two defenders play their own cards. Each of the three can see two sets of cards— their own and the one in the dummy. The difference in nature of their tasks arises because the declarer can (1) see the complete set of cards for her side, and (2) is the only one to make decisions about what cards to play based

on this information. Each of the defenders, on the other hand, has to play one set of cards without information about what their partner might hold. They can see the dummy's cards, but that does not give them enough information to plan the hand. The defender's task is generally considered to be more difficult. We take a brief look at declarer play, since not enough progress has been made in *Bridge* programming to discuss the complete game. Also, while *Computer Bridge* has been popular for a while, and there have been regular *Computer Bridge* championships, many of the programs like *Jack*¹⁸ and *BridgeBaron*¹⁹ are shrouded in commercial secrecy and are not available for study. Furthermore, the level of the computer programs is still not comparable to *Chess* vis-à-vis the human champions, and there seems to be still considerable scope for new ideas here.

On the face of it, the task of declarer play is like that of play in *Chess*, the declarer and two opponents selecting moves alternately. The major difference is in the fact that the declarer cannot see the cards each opponent holds. A direct consequence of this is that the game tree cannot be generated, and, therefore, algorithms based on *Minimax* search cannot be used.

One interesting approach to solve the declarer play problem is to adopt a Monte Carlo method. This approach was used in implementing the *GIB* program (Ginsberg, 1999, 2001). The main idea behind this approach is that if you knew the opponents' cards then the problem would become a complete-information game problem, known in bridge parlance as a *double-dummy* problem. A double-dummy bridge problem is still a difficult one to solve, generating a tree of about 10^{21} nodes (Khemani and Ramakrishna, 1989). But one could tackle it by using a heuristic approach in which not all moves are generated at each step, but only a few likely ones. The key idea behind the Monte Carlo is to generate a number of different hands for the opponents, and solve them using a double-dummy approach, and select the strategy that works in most of the generated hands. Furthermore, one could use the inferences from bidding to generate the hands that are consistent with the bidding to improve the results.

A second approach would be to treat the problem as a planning problem, and use as operators the high level *combination plays* that work under some conditions. An example is one of the simplest combination plays known as the *finesse*, which in one of its forms works as follows.

One pattern of finesse is applicable when the declarer has the Queen and the Jack in one hand, and an Ace and a small card in the other hand as shown below,

North: A 2

LHO

RHO

South: Q J

South can plan for two tricks by using the following combination play. Start play from the South hand with the Queen. If the *Left-Hand Opponent (LHO)* plays the King then play the Ace from the North hand, and win the next trick with the Jack. If LHO plays the small card, play the 2 from the North hand and hope that the RHO does not have the King. If that is the case, the trick will be won with the Queen. The next trick is won by the Ace.

In this manner, a finesse can be used to get two tricks in a situation when the second highest card is with the opponent. The finesse is a *conditional plan* that will succeed if the King is with the LHO and fail if the King is with the RHO. In the absence of any other inferences, the probability of the King being with the LHO is fifty percent, and therefore the probability of the plan succeeding is fifty percent. Employing knowledge of such combination plays, a planner can assemble different plans, and choose the one which has the highest probability of success. This was the approach used in (Khemani, 1989), and some of rules used in the implementation have been discussed in Chapter 6.

Such probability calculations figure prominently in bridge analysis, and the bridge literature too is replete with *Bridge* proverbs like in the world of Go. One of the proverbs says for example “*eight ever, nine never*”, which says that if the declarer has eight cards in a suit between the two hands, missing the Queen, she should always finesse against the Queen; but if the declarer has nine cards then she should never finesse. This is merely an easy way of remembering how the probabilities lie in the two situations. A declarer play program on the lines described above is essentially a kind of hill climbing approach in which the algorithm refines the most likely plan. It stores a list of assumptions needed for the plan to succeed, and if during play one of the assumptions were to be falsified, the program would need to plan again from that point onward.

Other later work is on similar lines, but more refined. The declarer play in the program *Tignum 2* (Smith et al., 1996) was an application of the *Hierarchical Task Network* (see Chapter 7) planning approaches developed by Dana Nau and his group. Figure 8.39 below depicts the task network for a finesse in *Tignum 2*. The position being planned for is when the south hand holds the king and jack of spades and hopes to gain an extra trick when East has the queen. The *Tignum 2* program is at the heart of the commercially available *BridgeBaron*²⁰ program.

Another interesting, and a much more complete treatment on bridge play is described in the doctoral thesis by Ian Frank (1998) that explored the relation between partial plans and proof planning. Their approach is motivated from the area of mathematical reasoning combined with explicit probabilistic reasoning with card combination. The details are beyond the scope of this book, and the interested reader is referred to the published work. As of writing this text, the champion program²¹ in the tournament circuit is called *Jack*²², but the author has no information on its working.

8.4.2 Scrabble

*Scrabble*²³ is a popular board game in which two or four players place letter tiles on a 15 by 15 board to form words according to rules similar to crossword rules. The game is played with a bag of 100 tiles, 98 of which are labelled with letters from the alphabet, along with a number between 0 and 10 representing the points associated with that letter. The other two tiles are blank and may represent any letter the player playing it chooses. Each player holds a *rack* of seven tiles drawn randomly from the bag of tiles, and these are hidden from the other player(s). The players play alternately, placing tiles on the board to form legal words as defined by some dictionary; for example the Official *Scrabble* Player's Dictionary (OSPD) published (Merriam-Webster, 2005), and replace the played tiles with new ones randomly chosen from the bag. Each new word formed should be meaningfully connected to the existing words, either extending one, or cross-linking with one, as in crosswords. At no time must there exist a sequence of letters not part of a word. After each move, the player gets a score which is the sum of the scores of the words formed by the player in that move. This score may be augmented when the tiles are placed on specially marked squares, known as hotspots, on the board. The marked square may double or triple of either the tile points or the entire word score. In addition, if a player uses up all the seven tiles on her rack, she gets a bonus of fifty points. This play is also known as a *bingo*. The game ends when there are no tiles in the bag, and either one player has finished her tiles, or no player can make a move. At the end of the game, each player's score is reduced by the sum of the points of their tiles left in the rack. In addition, if one player has finished all tiles, she gets the points of the tiles remaining on the other players' rack. The winner of the game is the player who scores the most points.

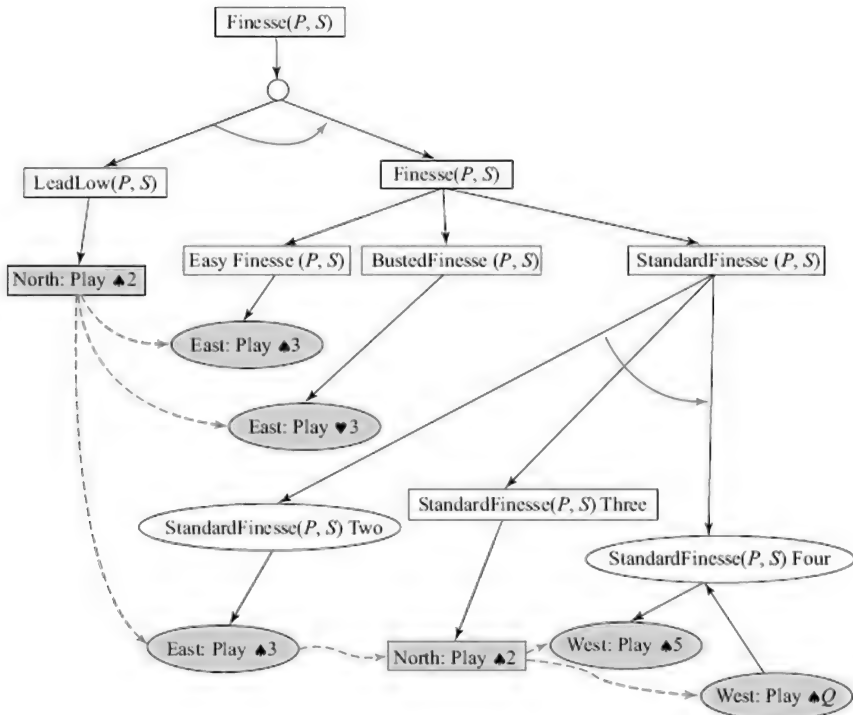


FIGURE 8.39 The portion of a task network for Tignum 2 adapted from (Ghallab et al., 2004) for a 'finesse' tactic in bridge. The strategy has the structure of an *And-Or* tree, where the *And* arcs point to the four cards played in that trick, and the *Or* arcs to the different opponent responses. The thick dotted arrows depict the sequence of plays. Observe the different notation. The opponent moves are represented in oval nodes, as opposed to states in which the opponent is to move.

Tournament *Scrabble* is played between two players, and we restrict our attention to the two player version. In the initial stages, the game is an incomplete information game because a player does not know what letters the opponent has in her rack, and, therefore, cannot predict the moves of the opponent. Further, since one cannot predict what new tiles one will draw from the bag, even your own moves at the third ply cannot be generated. Hence, the traditional methods based on the *Minimax* algorithm cannot be used. Moreover, a typical *Scrabble* position has about 700 possible moves (Sheppard, 2002), which may go up to 8000 if the player holds two blanks. Therefore, searching the game tree would anyway be difficult. In the end of the game, when the bag becomes empty, the game reduces to a complete information game and traditional methods can be used.

Obviously, performance in *Scrabble* is critically dependent of the vocabulary of the player. The more words a player knows, the more the possibility of making high scoring words on the board. In this respect, a computer program has a distinct advantage, since the entire dictionary can be stored. The OSPD contains 95,000 words, and this is

considerably more than the number of words humans typically have access to. Apart from the knowledge of words, there is also a combinatorial and tactical aspect to the game. Placing high point letters on hotspots yield more points. The letters left behind on the rack, known as *rack leave*, also have a bearing on the future scores of the players. In all these aspects, the computational power of a machine can be harnessed profitably, and it is no surprise that the best *Scrabble* players are computers. A brief history of early *Scrabble* playing programs can be found in (Appel and Jacobson, 1988). A program named *Maven*²⁴ developed by Brian Sheppard (2002) has long been the reigning champion. More recent work on modelling the opponent and guessing the tiles on her rack (Richards and Amir, 2007) has taken computer *Scrabble* performance even further. While *Maven* is now a commercial program, the program by Richards and Amir, *Inference Player*, was developed on top of a freely available open-source program²⁵ called *Quackle* (Katz-Brown and O'Laughlin, 2006). Richards' program makes inferences about the tiles left on the opponent's rack after their move. It infers that any letters that could have been used for high scoring words are not on the rack, because otherwise they could have been used. A similar inference, known as the *principle of restricted choice*, is made by bridge players about cards *not* played by an opponent.

We look at some of the techniques for implementing *Scrabble* as described by Sheppard (2002).

The Dictionary and Move Generation

In principle, the dictionary is a list of words. But searching a list of 95,000 words for legal words during each move is no mean task. One way to address this problem is to organise the words in a *trie* data structure. At every node, the children branch on the different possible letters to extend a partial word. For the English language, the root node will have 26 children, one each for words starting with the 26 letters of English. Nodes lower down in the tree structure would have fewer nodes. The node after the letter 'Q', for example, will have only one branch for the letter 'U', since in every word the letter is followed by the letter 'U'. If one combines the common suffixes of different words then the resulting network is a discrimination network known as *Directed Acyclic Word Graph (DAWG)*, and is a compact searchable structure containing the words in the dictionary (Aho et al., 1974). Figure 8.40 shows an example *DAWG* for a small collection of words.

Move generation for the game involves the following. The player must either identify whole words that can be placed on the board in such a way that it is connected to the existing words, or must identify a part of a word from her letters, such that the remaining letters are in place on the board already. In both cases, one must ensure that the move does not result in any sequence of letters on the board that is not a legal word. Move

generation is thus a critical part of a *Scrabble* program.

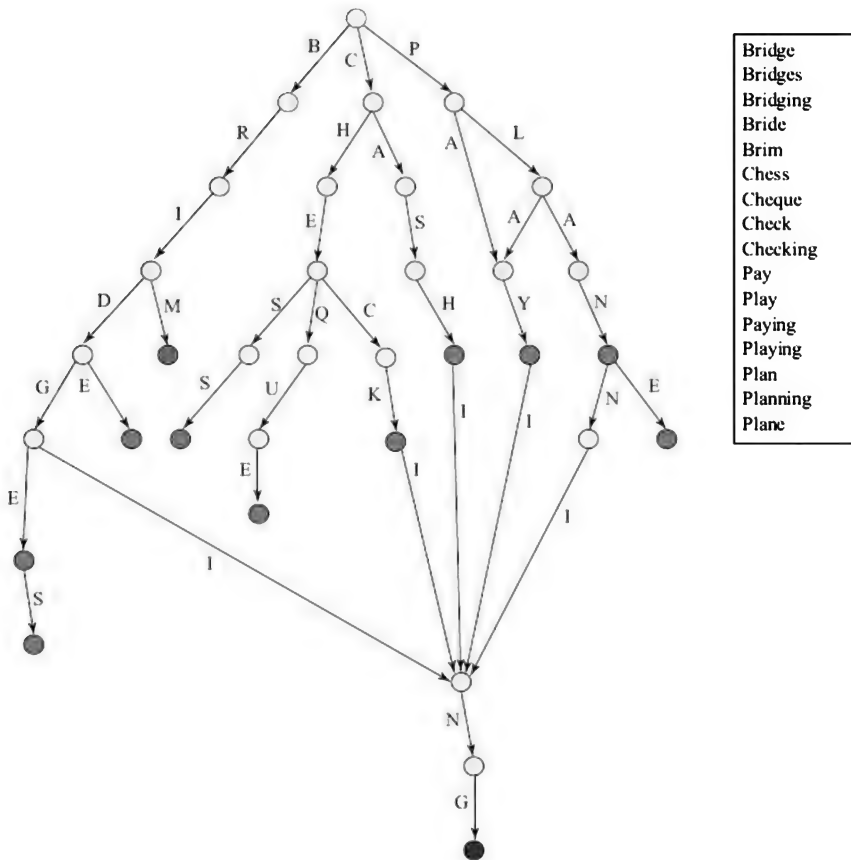


FIGURE 8.40 A sample trie data structure for the set of words in the box on the right. The darker nodes represent legal words.

Appel and Jacobson (1988) use the *DAWG* structure to implement a fast move generator. Their algorithm first scans the current board looking for places where a word might be formed connecting to existing letters. Then it attempts to build words using letters on the rack and the letters near the chosen spot. Before searching for words, the program computes a *cross-check* for every existing word. If the program is searching for horizontal words then it precomputes what letters are feasible in the squares above and below an existing down word. For example, as shown in Figure 8.41, if a horizontal word is to be placed occupying one of the squares marked '?' then one could use a 'P' on the upper square or an 'R' in the lower one while making the new word. The set of allowable letters can be represented as a 26-bit vector, with one bit for each letter in the alphabet. Changes in these values will be few during each move of the game. In addition, the program precomputes *anchor* squares on the

board where a move may be made. Anchor squares are empty squares next to existing letters where a word may be made. Note that squares marked '?' in the figure are also anchor squares.

As described in (Appel and Jacobson, 1988): "*The move generation problem is thus reduced to the following one-dimensional problem: given a rack of tiles, the contents of a row on the board, and the cross-checks and anchors for the row generate all legal plays in that row.*"

Each word is generated in two phases. In the first phase, the *left part* of the word is generated. The left part is the part of the word that is to the left of the anchor. Observe that the left part of a word can be empty, for example when a word begins in the square marked '?' in Figure 8.41. It is also possible that the left part may already exist on the board, for example for the blank square in the above figure. Again, one might note that the left part could be longer than one letter in this case. After this phase, either the left part of the word exists, or if it is empty, a constraint on the anchor letter exists in the form of the cross-check constraint. Thus, the right part of the word is generated as a constrained extension of the left part. The generation algorithm of Appel and Jacobson essentially tries out all feasible combinations of words that can be generated in the two-phase manner described above. The other constraints on the words generated are the constituent letters must be available, either on the board in the left part, or in the rack held by the player. In addition, the trial and error is not brute force. It is constrained to the paths available in the *DAWG*. Only meaningful letter permutations need be explored. For example, if the generator is exploring the anchor next to the letter 'T' in the above figure, only paths starting with letter 'T' are explored in the *DAWG*, and only those paths are explored for which the player has the letters in the rack. The detailed algorithm and issues are available in the paper. The algorithm was very successful and was also incorporated into many other *Scrabble* programs, including *Maven*. The *DAWG* is conceptually similar to the *Rete net* we studied in Chapter 6, in the sense that only desired matches are done. A word is like the left-hand side of a rule. In this case, it also behaves like a finite state machine whose accepting states are nodes ending in meaningful words.

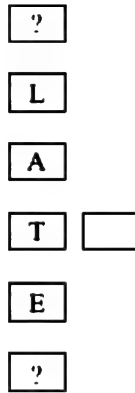


FIGURE 8.41 A cross-check precomputes what letters can be placed meaningfully in the squares marked '?'. An anchor is an empty square adjacent to an existing letter where a word can be placed. In the figure, a word starting with T may be made at the empty square shown next to the letter 'T'.

One problem with the above algorithm is that it constructs words from left to right. In *Scrabble*, however, one needs to hook a new word from any of its letters that satisfies the cross-check constraints shown in the above figure on the square marked '?'. For example, if a player has the letter 'R' then she could think of a horizontal word below 'LATE' extending it to 'LATER'. The only constraint on the new word is that it should have an 'R' in it. Appel and Jacobson's algorithm is forced to try all combinations of words from left to right, because the *DAWG* is organised in this fashion.

In a new algorithm (Gordon, 1994), the *DAWG* structure is replaced by another structure called *GADDAG*, which is bidirectional in nature²⁶. Unlike the *DAWG*, which allows one to search only from the first letter, the *GADDAG* allows one to search in both directions from a letter anywhere in the word. The size of the *GADDAG* is reported to be about five times the *DAWG*, but it generates words twice as fast. In a world of expanding computer memories, this is perhaps a good trade-off, specially if time is going to be of essence, as it is in a tournament. We will not discuss the details here, but the interested reader is referred to the paper by Gordon.

Play

The game is characterized by the following properties,

- The branching factor is high, ranging from 700 without blank tiles, to as high as 8000 when a player holds two blanks (Sheppard, 2002).
- The opponent moves cannot be generated since the opponent's rack is hidden. This feature of incomplete information is similar to the one in card games like bridge. As a consequence, the second ply cannot be generated.

- Even the player's next move cannot be generated. This is because (1) the opponent's move is not known, and (2) because the tiles that will be drawn by the player are not known. The second feature is different from games like *Bridge*.

Maven divides the game into three parts, each of which it treats differently.

The first part is the initial game or the normal game, governed by the above properties. *Maven* plays this phase using a one-ply search. That is, it generates all moves and chooses the best looking one. Everything thus depends upon the evaluation function. The evaluation function has the following components.

Rack Leave A very important feature is the tiles left behind on the rack, known as the rack leave. Obviously some letters, for example 'E' and 'T' are easy to use because they occur in many words, while letters like 'Q' and 'Z' are more difficult to use. The rack leave has an impact on the score in future moves, and players would not like to leave bad combinations of letters. Thus, they have to trade off current score versus future score. *Maven* used games played against itself to learn this part of the evaluation function (see Chapter 18 on Machine Learning). It tracks individual tiles and combinations of tiles like duplicated letters, and vowel-consonant combinations. It learned values for these parameters, by completing games with different rack leaves, and recording the future score associated with each combination. The future score is the difference in the score of the side to move and the opponent over the rest of the game. It collected these scores over a set of games, and used these to tune the values associated with the letter combinations in the rack leave.

Current Score The points scored by the current move obviously contribute to the final score. Thus, this is a direct measure of the goodness of a move.

Board Position Surprisingly, the board position does not contribute much to the evaluation function. This is contrary to the opinion of experts that giving openings to the opponents is not a good idea, and one should be willing to sacrifice a few points to avoid opening up the board for opponents. Experiments reported with *Maven* reveal the opposite; that board position does not contribute to the evaluation function. Intuitively, one can understand this by observing that the board in *Scrabble* is fundamentally different from the board in other games. It is the same for both sides! In games like *Chess*, *Go*, *Backgammon* and *Checkers*, the two players have their own coins, and thus the arrangements of the coins are a reflection of the status of the game. In *Scrabble*, on the other hand, the coins are not owned by any one player, and can be used in future moves by both. In that sense, an open board is equally good for both. That only leaves the question of the immediate move advantage.

Experiments with *Maven* reveal that this is negligible, except in the case of triple-word openings. This is when an opponent can gain a significant advantage. There too, the value of the opening depends upon the state of a game. For example, playing a 'Q' near the triple-word hotspot may not yield value in most positions, specially in the end game. *Maven* has a table of values of letter combinations near triple-word that it uses to evaluate the cost of giving such an opening.

Simulation Even though one cannot generate the game tree, one can generate a plausible game tree by assuming some tiles for the opponent and for the draw, and compute the score at the end of some amount of look-ahead. If this is done a sufficient number of times then one can choose a move which on the average yields better results. Such a Monte Carlo approach was also used in *Bridge* in the program GIB which generated plausible hands for opponents and played out complete information games (Ginsberg, 1999). It was also used successfully in the game of *Backgammon* described below, where it was called a *rollout*. This approach becomes more feasible in the second phase described below.

The second phase, or the pre-endgame stage (Sheppard, 2002), begins when there are 16 unseen tiles, 7 on the opponents rack and the rest in the bag. In this stage, some look-ahead can be done based on some educated guesses about what the opponent holds, and some informed *fishing*²⁷ for some desired letters. This is where the program *Inference Player* (Richards and Amir, 2007) would gain advantage over programs that have no idea about what tiles are in the bag and which ones on the opponent's rack.

The third stage, the end game proper, begins when the bag is empty. At this stage, both players know what tiles the other player has, and the game turns into a complete information board game. The branching factor is still high, and *Maven* resorts to the B^* algorithm (Berliner, 1979) that does selective search over a game tree.

8.4.3 Backgammon

One of the earliest game playing programs to make an impact at the highest level was the game of *Backgammon* (Berliner, 1980). It was a time when *Chess* programs were making progress, but were not quite at the level of human champions. Of course, Samuel's checkers playing program (Samuel, 1959) had made an impact earlier, by demonstrating that a program could learn and play better and better. Subsequently, a program named Chinook (Schaeffer et al., 1992) (Schaeffer, 1997) did become the world champion²⁸ in 1994. But Checkers does not evoke so much passion as do some of the other games we have been discussing. Furthermore, the authors of Chinook have announced (Chang, 2007) that the program cannot be beaten. Like the children's game of *Noughts and*

Crosses, the game of *Checkers* is also now solved. Curiously, Samuel's *Checkers* program and the reigning *Backgammon* programs share the same methodology of learning from playing against themselves.

Backgammon is amongst the oldest known games, reputedly being played a thousand years before *Chess* (Tesauro, 1995). Certainly, a similar game called *Chausar*²⁹ has been described in ancient Indian literature (Handelman, 1997). A particular incident of the game from the Indian epic *Mahabharata* (traditionally ascribed to the sage Vyasa; many translations are available) is ingrained in the Indian psyche because it involved the gambling away of *everything*³⁰, eventually leading to the epic battle.

Modern *Backgammon* is played on a board of 24 locations called *points*³¹ on which two players start with 15 *checkers* (or pieces) as shown in Figure 8.42. The 24 points are partitioned into four quadrants as shown in the figure. The objective of the game is to bring one's checkers to one's home quadrant, and then move, or *bear*, them off the board. The moves are dictated by a pair of dice. For example, the dice in the figure show 4 and 1, and this means that the player can move one checker 4 steps and another, possibly the same one, 1 step. Every move can only be made to an *open* point that contains one or zero opponent checkers. If a player rolls double (same number on both dice) then the player gets four (double) moves. The players move in opposite directions from a symmetric starting position, and the first one to bear all checkers off, is the winner. In addition, if the losing player has all checkers on board, the winner wins a *gammon*. If the loser has a checker in the winner's home or on the *bar* (see figure) then she loses a *Backgammon*. The players may also raise the stake by using a doubling dice that is labelled with the numbers 2, 4, 8, 16, 32 and 64.

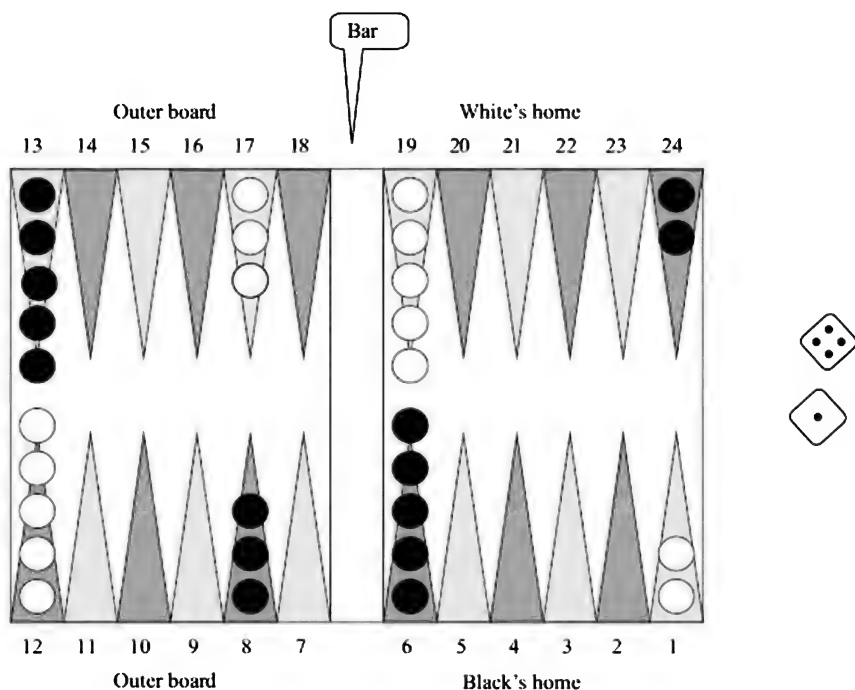


FIGURE 8.42 The initial position in *Backgammon*. The numbers are shown for Black. The objective is to bring your pieces to your home, and then “bear” them off the board. The moves are decided by a pair of dice.

Thus basically, the game is a race between two players, running in opposite directions on the same track. The speed of the two players is determined by the throw of the dice. The game gets its complexity from the interaction between the checkers of the two players. For one, a player cannot place her checker on a point occupied by two or more of the opponent’s checkers. As a corollary, players can aim to block the opponent by occupying crucial points with two or more checkers. Secondly, a checker can be played on a point that has one checker of the opponent (known as a *blot*). In doing so, the resident checker is *hit*, and has to move to the bar (shown in the figure) and try and get back on the board again.

A preliminary throw of a single dice decides who plays first. After that, the players play alternately until one of them has moved all the checkers off the board. The decision making task of the player is, given the throw of a pair of dice, to choose from the set of moves allowed by the throw. One can think of the dice throw as randomly pruning the branching factor at each level, keeping only moves determined by the outcome of the throw.

Traditional *minimax* based techniques cannot be used because one does not know the outcome of the throw by the opponent. Searching through all possibilities is also not feasible because there are 21 different

outcomes of a throw of two dice, and each outcome typically allows around 20 moves. So the branching factor would be about 400, much larger than the game of *Chess*, and more like the branching factor in the game of *Go*. Not surprisingly, the techniques of *Backgammon* also rely heavily on evaluation.

Two of the most successful *Backgammon* programs written by Gerald Tesauro illustrate two very different methods of *learning* the evaluation function.

The first program called *Neurogammon* (Tesauro and Sejnowski, 1989) used a feed-forward neural network (see Figure 4.20) to learn the mapping between board positions and moves. Its input representation included both the raw board information (number of checkers at each location), as well as a few handcrafted “features” that encoded important expert concepts, plus a final position after the move. The output was a value of the move in the range $[-100, 100]$. The program was trained by using the *Backpropagation* algorithm (Werber, 1994) on a set of recorded expert games. This is an example of supervised learning, in which the system is shown a set of desired values by an expert, and essentially learns a nonlinear function approximation representing the mapping between the input output pairs. *Neurogammon* was very successful and won the *Backgammon* championship at the 1989 International Computer Olympiad quite easily (Tesauro, 1989).

The second, and much better, program by Tesauro (1994, 1995, 2002), called *TD-Gammon*, learnt the evaluation function in an entirely different manner. While *Neurogammon* got its knowledge by being taught by experts, *TD-Gammon* learnt the principles of *Backgammon* strategy by itself, playing 300, 000 games against itself and learning the good moves, or *afterstates*, as they are known in the research field of *reinforcement learning* (Sutton and Barto, 1998).

The specific form of learning that the program did is called *Temporal Difference* (TD) learning, and hence its name. TD learning addresses the problem of *credit assignment* to moves when the reward is only available at the end of a sequence of actions. When a player plays a game, the result is known only when the game ends. The task is to decide which of the moves contributed by how much to the result. The algorithm used in *TD-Gammon* is the algorithm known as *TD(lambda)* (Sutton, 1988). The training, as described in (Tesauro, 1994), is as follows.

The complete set of moves played by the program is fed to the feed-forward neural network. The board positions are the input vectors $x[1]$, $x[2]$... $x[f]$ to the network. For each input pattern $x[t]$, there is a neural network output vector $Y[t]$ indicating the neural network’s estimate of expected outcome for pattern $x[t]$. $Y[t]$ is a four-component vector corresponding to the four possible outcomes of either White or Black, winning either a normal win or a gammon. At each time step, the *TD(lambda)* algorithm is applied to change the network weights, using the following formula,

$$w_{t+1} - w_t = \alpha(Y_{t+1} - Y_t) \sum_{k=1}^t \lambda^{t-k} \nabla_w Y_k$$

where α is a small constant, commonly thought of as a “learning rate” parameter, ‘ w ’ is the vector of weights that parameterizes the network, and $\nabla_w Y_k$ is the gradient of network output with respect to weights. The equation computes the weight change for a single output unit. For multiple units, the computation is repeated. The quantity λ is a heuristic that controls the degree of temporal credit assignment. When $\lambda = 1$ then the feedback goes arbitrarily back in time. That is, it affects the weights of all preceding moves. At the other extreme, when $\lambda = 0$, the no preceding moves are affected. With different values in between, one can control the extent to which the weights are affected. Tesauro used a value of 0.7 initially, but later used the value 0 since the performance was similar but much faster (Tesauro, 2002).

The first version of the program used only a raw representation of the board. A total of 198 input nodes represented the position of the checkers, at each point and on the bar, and the number borne off the board, and whose turn it is to move. It did not have any knowledge of *Backgammon* encoded by an expert. With 40 internal nodes and 200,000 training games, it was able to compete successfully against *Neurogammon*!

In the second version of *TD-Gammon* 1.0, Tesauro combined the knowledge intensive handcrafted features of *Neurogammon* with the unsupervised TD learning, and found that the program was a much better player. It was trained over 300,000 games. Next, in *TD-Gammon* 2.0, a *selective* two-ply search was introduced. The second ply search was done only on a subset of (good) successors at the one-ply level, determined by applying the evaluation function. Then in the second ply, a 1-ply move decision is made for *each* of the 21 possible dice rolls for the opponent, and the probability weighted average (weighting nondoubles, twice as much as doubles) of the resulting states is computed. The two-ply search process is illustrated in Figure 8.43 below.

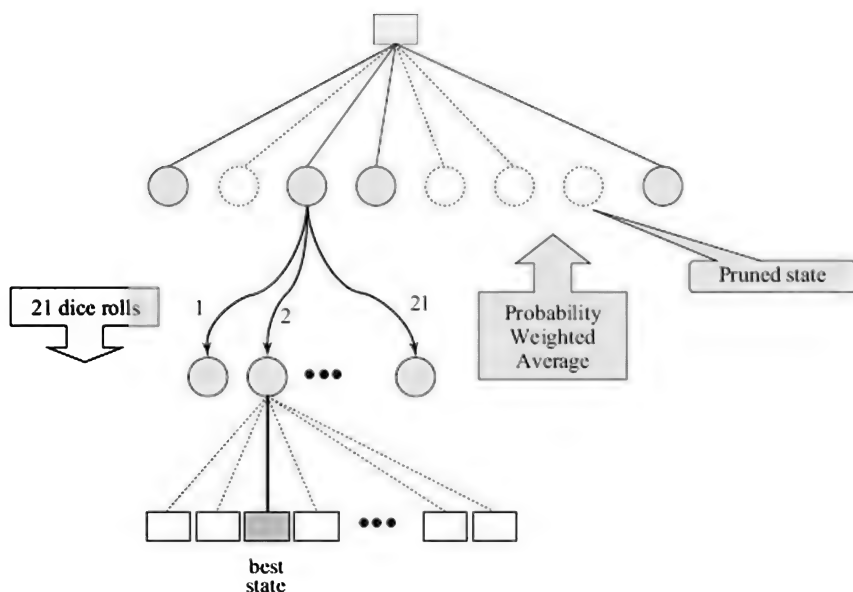


FIGURE 8.43 The two-ply search in *TD-Gammon 2.0*. At one ply, some of the states are pruned. Then for each of the remaining states, the best move for each dice throw is picked by the second ply search. The values of the best states are averaged based on probability. The figure shows the process for one selected node at the first ply.

TD-Gammon 2.0 and *TD-Gammon 2.1* were trained over 300,000 and 1,500,000 self-games respectively and were comparable in performance to the best human players in 1992 (Robertie, 1992). *TD-Gammon 3.0* had a three-ply search done as follows. The first ply was done as in the two-ply search, but at the next level, another 2-ply search was done. For each of the 'best move' selected in the second ply, a further 21 dice rolls were done and the weighted average computed in a similar manner. Thus, each value at the first ply level was backed up from 441 nodes. *TD-Gammon 3.0* had 160 hidden units and was trained for over 6 million self-play games (Tesauro, 2002), and averaged 10–12 seconds per move decision in 1998, running on a 400 MHz Pentium-II processor.

Evaluating *Backgammon* games is not easy because whether a move turns out to be good or not may be dependent on future dice throws. Such a problem also occurs in *Scrabble*³² because the future score may depend upon the letters drawn from the bag. And playing a large number of games against humans for evaluation is not feasible. *TD-Gammon* introduced a different way of evaluating moves. That is the technique of *rollout*. A rollout is basically playing many game continuations from a given position, each time using randomly generated dice throws. The rollout score of a move is the average score obtained by the continuations commencing with that move. The move with the highest rollout score is judged to be the best move. And because the computer can play a very large number of games, the results have proven to be reliable; so much so that computer rollouts have become the standard

method of evaluating *Backgammon* moves (Woolsey, 2000; Montgomery, 2000). Tesauro has suggested that with increasing computing power, one will be able to use the Monte Carlo rollout methods to produce much better programs. This approach also forms the basis of one of the successful *Bridge* playing programs (Ginsberg, 1999, 2001) and in the end game in *Scrabble* (Sheppard, 2002). Sheppard did try and use TD learning for the entire game of *Scrabble* but without success. He suggests that the algorithm works with spectacular results in *Backgammon* because the dice throws produce a sufficient number of states to cover the entire space; while in the case of *Scrabble*, the approach gets caught in local optima.

The two approaches used in *Neurogammon* and *TD-Gammon* are both learning systems in which neural networks are trained to evaluate positions. But they differ widely in the source of learning. *Neurogammon* learnt from moves made by experts and as a consequence its knowledge was a reflection of the experts' knowledge. *TD-Gammon*, on the other hand, started from scratch and learned by playing a large number of games against itself, rewarding moves that led to winning positions. It quickly developed the basic concepts in the game, and went on to become a champion level player. Its learning was unsupervised in contrast to the supervised learning of *Neurogammon*. In the process, a curious thing happened. Many of the notions that human experts held were proven to be wrong, eventually leading to modification of the expert knowledge itself. An example given in (Tesauro, 1994) illustrates such a change. In the opening position, if Black was to roll 4–1 then the expert opinion was to play [13–9, 6–5]³³ but was changed to [13–9, 24–23] after *TD-Gammon* preferred it, and was subsequently validated by rollout. In his way, *TD-Gammon* has altered the way in which human experts evaluate positions!

8.5 Beyond Search

There are some games that can be analysed in advance completely. Consider, for example, the game, which we will call *Pic123*, where there are a number of coins (or sticks) on the board, and a player can pick up one, or two, or three coins at a time. The last one to pick a coin, and clear the board, is the loser. Obviously, a state with only one coin is unsafe because the player has to pick that coin, and clear the board. One can see that if there are two or three or four coins on the board, a player, say *MAX*, is “safe”. *MAX* can pick up an appropriate number leaving only one coin on the board, which then *MIN* is forced to pick. Working backwards, one can see that if *MIN* is looking at five coins then whatever *MIN* does results in a safe state (2, 3 or 4 coins) for *MAX*. So a state with five coins is also unsafe, provided *MAX* plays perfectly. We can extend this argument and see that the states can be categorized as follows.

If a state has $(1 + 4n)$ coins, it is unsafe, where n is a natural number. Else, the state is safe.

The only thing that one needs to do to win this game is to ensure that the opponent is in an unsafe state. Then whatever the opponent does, one can put her back in an unsafe state, as shown in Figure 8.44 below. Observe that the graph in the figure is a strategy, containing one move for MAX and all for MIN. And this strategy is a winning strategy. Of course, if the opponent has also done this analysis then the starting board position completely decides the outcome, *without having to search* the consequent game tree. One only needs to count the coins on the board, and the game is no longer of any interest. Observe, that with a large number like 400, the game tree would have a depth of 400 moves with an average depth being about 200 moves. With a branching factor of three, this is quite a big tree. But faced with the knowledge gleaned from the above analysis, the game can simply be played in constant time!

The kind of analysis done for above is not only restricted to games like *Pic123*. Even in other board games, one can analyse positions in advance and prepare a generalised strategy that can be executed without going through search. This is, in fact, done in games like *Chess* as a matter of routine, but only for certain end positions. For example, most players quickly learn how to mate the opponent's king with a rook and a king, with two rooks, with a queen and a king, with two bishops, etc. The game subtree below these positions may be quite large, but seasoned players tend to treat such positions as terminal nodes, just like we did in the *Pic123* game above.

This means that playing the game is no longer just using search to look ahead, but also collecting and applying a set of specialized methods for specific situations which have to be recognized by some kind of pattern recognition process. This involves more sophisticated knowledge representation and processing techniques. Because search methods are general in nature and do not require special treatments for special situations, they may perform poorly in situations where specialized knowledge would solve the problem quickly. This is the case, for example, when a *Chess* player uses triangulation to achieve *Zugzwang*, or uses a special procedure to mate the opponent's king with two bishops, or a *Go* master employs a *Yose* in the end game. While such specialized procedures work very well when they are applicable, and recognizing their applicability is a task in itself, they work only in those situations. Knowledge based methods thus may suffer from a problem of incompleteness, in the sense that they may not have a solution in all situations. One of the key open problems in artificial intelligence is to devise integrated representations that can be used by both kinds of methods, and implement hybrid multipronged problem solvers that may choose an appropriate procedure in different situations.

8.6 Discussion

Games have always fascinated us. They have been means of recreation and have been considered a hallmark of intelligence. They provide us a means to pit our wits against others and have been the training ground for learning strategy and analysis. Because they are by nature symbolic or digital board games can be implemented on computers without any loss by abstraction. Furthermore, since rules and outcomes are well defined, the result of combat can be evaluated precisely.

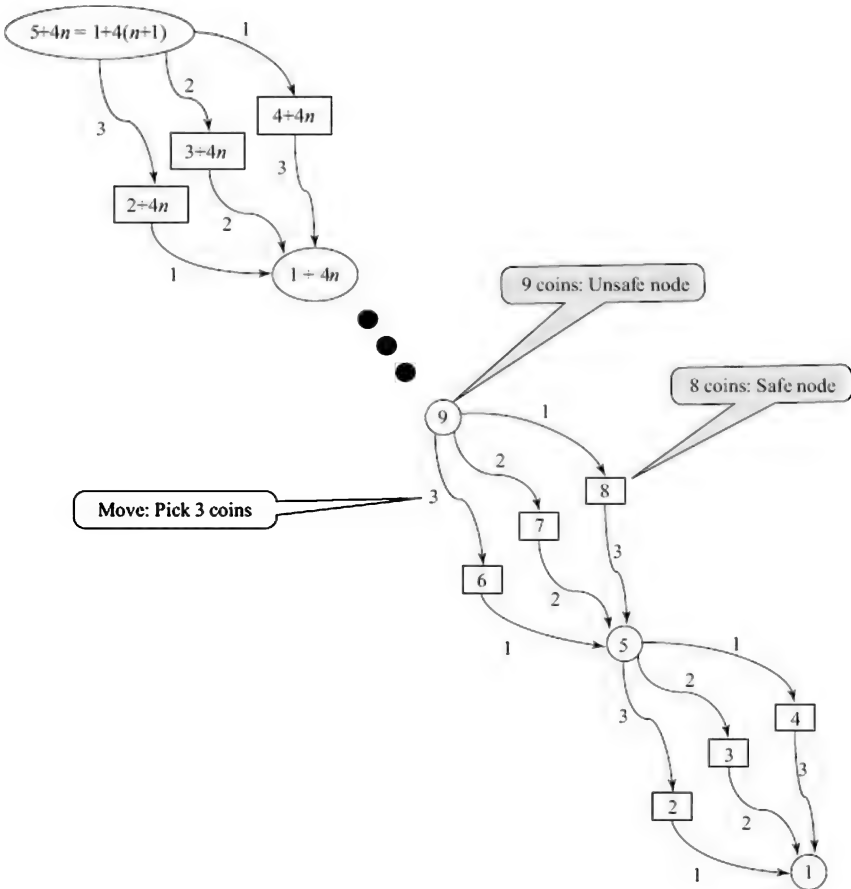


FIGURE 8.44 States with $1+4n$ coins are unsafe in the Pic123 game. These are shown in circles in the above figure. A player in an unsafe state can be forced to move to the lower unsafe state, and eventually to the losing state, where there is only one coin on the board.

Games like *Chess*, *Checkers* and *Go* are the simplest conceptually. But their complexity is still high enough to bar complete analysis and bring in the role of judgment in the form of evaluation. A complete analysis of *Checkers* was done in 2007 though. “*Perfect play by both*

sides leads to a draw.” (Schaeffer, 2007). A combination of evaluation and look-ahead has been good enough to implement programs to best human champions in *Checkers* and *Chess*, but *Go* is still beyond the pale of these techniques. These games have demonstrated the limits of search, even in these small finite worlds.

Scrabble and *Backgammon* are two of the most popular games that are different from the complete information games like *Chess* and *Go*. Both these games have had the distinction of computer implementations that are better than human players. Both rely on machine learning methods to learn evaluation of positions. *Scrabble* also benefits from the large vocabulary that a program can have access to. Both benefit from being able to search many possibilities very quickly. While both rely on the knowledge of evaluation, *Backgammon* has also demonstrated that human knowledge, even of the experts, can be fallible. Computer implementations of games like *Backgammon* and *Othello* have changed the way humans play these games.

Bridge is still a challenge. This is probably because this is a multi-player, two-stage game. A player, by herself, cannot decide upon a winning strategy. The best moves have to be found in a concerted manner by the two players in the partnership. This involves planning for formal communication, understanding, interception and deception that contribute to the decision making for the play itself. It requires the ability to make inferences and to do probabilistic reasoning. It probably offers a domain for the integration of various artificial intelligence techniques.

There are many other games that have evinced interest from enthusiasts and AI aficionados. A special issue of the journal *Artificial Intelligence* (volume 134, numbers 1–2, 2002) gives a wider selection. We have only looked at a few of them to explore the general principles that can be used to implement them. The interested reader will find many avenues for exciting challenges.



Exercises

1. The game *Undercut* consists of a sequence of moves in which two players simultaneously choose an integer between one and five, both inclusive. Each person gets the number she chooses as her score for that round, except when the opponent has chosen a number smaller than hers by one, in which case the opponent gets both the numbers. For example, if *A* chooses 5 and *B* chooses 3 then *A* gets 5 and *B* gets 3. But if *A* chooses 5 and *B* chooses 4, *A* gets nothing and *B* gets 9. Devise a strategy to play the game. For variations on the game, see (Hofstadter, 1996).
2. There are two companies producing a product of no intrinsic value and where sales are proportional to advertising. Advertising,

however, costs money and cuts into the profits. Another way of increasing sales is by reducing the selling price, though this may lead to a price war. Model the price war between the two companies as a two-person game.

3. Let G be the set of *all* finite games. We define the game *gnew* as follows. Player-1 selects a game, say game- i from the set G . Player-2 makes the first move in game- i , and they play the game till completion. Is *gnew* a finite game?
4. Modify the value of only one leaf node to convert the game in Figure 8.6 to one where the *minimax* value is a draw (D).
5. What is the size of the game tree for *Noughts and Crosses*?
6. "If you know the strategy that an opponent is employing, you can predict all her moves accurately." Is the preceding statement true or false, in context of the game of chess? Justify your answer.
7. Modify the algorithm given in Figure 8.17 to keep track of depth. Write the terminal(node) function.
8. Modify the algorithms given in Figures 8.17 and Figure 8.18 so that they also return the set of moves available to *MAX* after two plies (for each possible node) along with their backed-up values.
9. Modify the *AlphaBeta* algorithm to use the backed-up values stored in the above question, to order the moves of the algorithm in the next cycle.
10. In Section 8.2.2, we have observed that $a < V(J) < b$ where,

$$\alpha = \max \{\alpha_1, \alpha_2, \alpha_3, \dots\}, \text{ and}$$

$$\beta = \min \{\beta_1, \beta_2, \beta_3, \dots\}$$

Argue that the following condition holds for $V(J)$ to influence the root node.

$$\alpha_1 < \alpha_2 < \alpha_3 < \dots < V(J) < \dots < \beta_3 < \beta_2 < \beta_1$$

11. Will the *AlphaBeta* algorithm ever yield an inferior solution as compared to the *Minimax* algorithm? Give reasons for your answer.
12. Starting with a randomly chosen number, construct a 4-ply binary game tree with no cutoffs when explored by the *AlphaBeta* algorithm. Reverse the tree, and try the *AlphaBeta* algorithm again on it.
13. Construct a 4-ply binary game tree, using only the values 0 and 1 for the leaf nodes, such that there are no cutoffs with the *AlphaBeta* algorithm searching from left to right.
14. Show how the algorithm *AlphaBeta* explores the game tree, searching from left to right.
 - (a) Fill in the leaves that are inspected by *AlphaBeta*.
 - (b) Show the cutoffs and label them with their type.
 - (c) Mark the move that *AlphaBeta* will choose for *MAX* at the root.

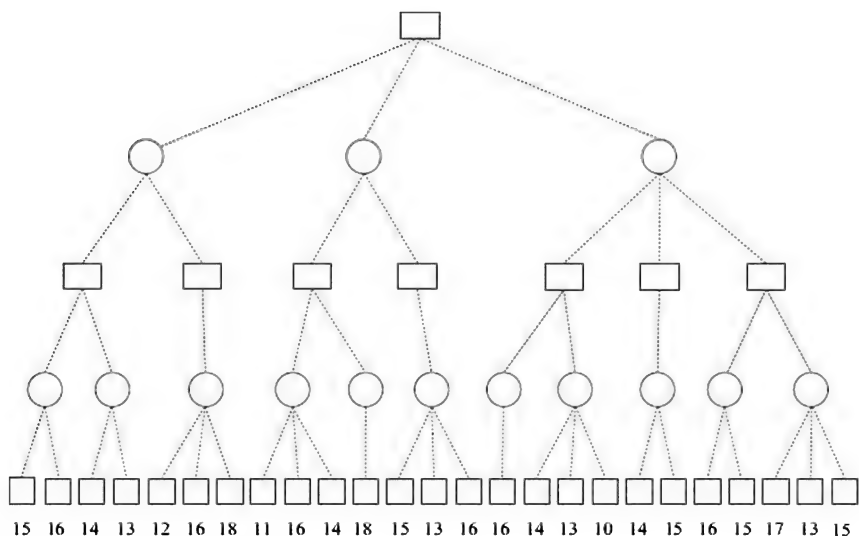


FIGURE 8.45 A small game tree.

15. Would the *Minimax* algorithm have chosen the same move? Give reasons.
16. In the game tree on the following page, the leaves are labelled with the values from the evaluation function. The letter labels [A ... X] below the leaves are names of the leaves. Show the order in which algorithm SSS* will inspect the nodes, explaining all the decisions made, along with diagrams where appropriate. What is the minimax value of the game?

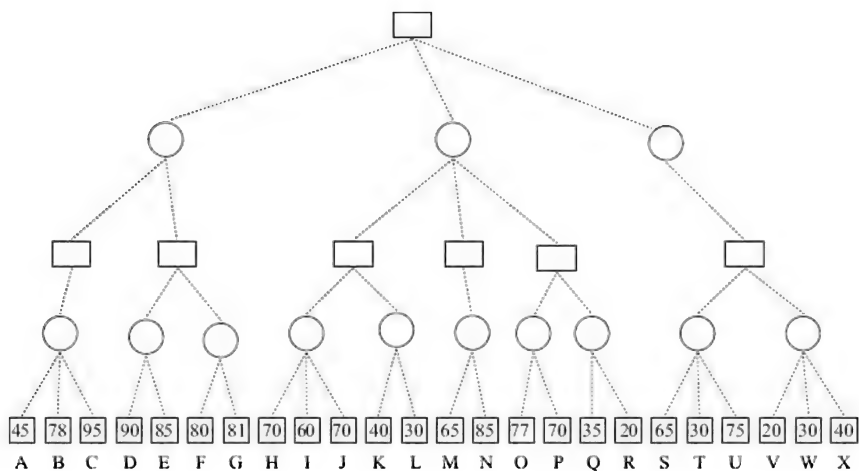


FIGURE 8.46 Another small game tree.

17. Compare and contrast the SSS* algorithm with AO* algorithm.

18. (For Chess enthusiasts) Is the Chess game tree a finite tree or an infinite tree? Can the two players keep repeating some inconsequential moves indefinitely?
19. (For Chess enthusiasts) Consider the following variation on Chess rules. In addition to the other rules a player is allowed to move a piece X onto a square containing his own piece Y. The two pieces then combine to form one piece XY with the mobility of both. It is claimed³⁴ that with these rules there is a strategy for a player to ensure that game will always end in a draw. Comment.

¹ “According to popular legends, the chess game was invented by a Brahmin named Sissa. One day the Indian king (rajah) Balhait summoned Sissa and requested the wise man to create a game which would require pure mental skill and oppose the teaching of games in which fate (luck) decides the outcome by the throw of dice. Moreover, the king requested that this new game should also have the ability to enhance the mental qualities of prudence, foresight, valour, judgment, endurance, circumspection, and analytical and reasoning ability. Sissa created the game chaturanga. Chaturanga was a war game, the first to borrow explicitly and extensively from the vocabulary of military conflict. The pieces were the king (rajah), the general (senapati/mantri, or modern day queen), the elephant (gaja, or the modern day bishop), the cavalry or horse (ashwa, modern day knight), the chariot (ratha/sakata, or modern day rook), and finally, the infantry (sainika/bhata, or modern day pawns)” – from *Wikipedia* (http://en.wikipedia.org/wiki/History_of_chess).

² This sort of exemplifies the attitude of AI detractors: that if the computer can do it, it does not characterize intelligence.

³ A tree structure in which each branch is a decision.

⁴ More recent studies have shown that such rationality exists only in the abstract scenarios. In real life, human psychology plays an important role, as depicted in many a movie about a pair of characters stepping outside the law. One might also observe that in the real world, not all countries have police forces that leave room for such rationality games.

⁵ We will often refer to the other player as an opponent for the sake of brevity.

⁶ Sometimes the term ‘game’ is used to refer to a *strategy* or a particular deviant kind of *behaviour* adopted by an agent, as described in the book “Games People Play” by Eric Berne. We are not interested in games in that sense, but as platforms on which agents act.

⁷ For example, Michael Howard in his book on strategic deception in the Second World War (Howard, 1995) describes how a nonexistent U.S. Army group pinned down an entire German Army

in the Pas de Calais, until Montgomery's forces had achieved a secure foothold in Normandy.

⁸ Recent literature also uses a triangle pointing up ▲ for *MAX* and a triangle pointing down ▼ for *MIN*.

⁹ We assume that the reader is familiar with *Chess*, or has access to one of the many introductory books on the subject.

¹⁰ "*Impossible is a word found only in the dictionary of fools.*" – Napoleon Bonaparte.

¹¹ The actual implementation is not recursive and in fact, at this point SSS* shifts attention to cluster *A*, as shown in the hand simulation below.

¹² http://en.wikipedia.org/wiki/Hans_Berliner.

¹³ In practice, *Hitech* uses a fairly complicated evaluation using chess specific knowledge about moves that are checks, that threaten mate, or that start with certain kinds of captures.

¹⁴ "It was named after *Deep Thought*, a fictional computer in Douglas Adams' series, *The Hitchhiker's Guide to the Galaxy*. The naming of chess computers has continued in this vein with Deep Blue, Deep Fritz, Deep Junior, etc. ("Deep" here generally refers to the special ability to use multiple processing units.) In 1994, *Deep Thought II* won the North American Computer Chess Championship for the fifth time, with its rating estimated at around 2600. It was sponsored by IBM. Some engineers who designed *Deep Thought*, also worked in the design of *Deep Thought II*. Its algorithms were quite simple evaluation functions, but it could examine half a billion chess positions per move in tournament games, which is sufficient to reach a depth of 10 or 11 moves ahead in complex positions. Despite that, using the technique of *singular extensions*, it could also follow lines of forced moves that reach even further, which is how it once found checkmate in 37 moves." – from

http://en.wikipedia.org/wiki/Deep_Thought_%28chess_computer%29

¹⁵ Rules of Go, Wikipedia, http://en.wikipedia.org/wiki/Rules_of_Go

¹⁶ Actually, the number in the bid is six less than the number of tricks bid for. The lowest bid is 1, for seven tricks, and the highest 7, for thirteen tricks.

¹⁷ Many a hilarious incident has occurred on the bridge table because the partner has forgotten the encoded meaning and taken the bid at face value.

¹⁸ <http://www.jackbridge.com/eindex.htm>

¹⁹ <http://www.bridgebaron.com/home.shtml>

²⁰ <http://www.bridgebaron.com/home.shtml>

²¹ <http://www.ny-bridge.com/allevy/computerbridge/results2007.html>

²² <http://www.jackbridge.com/eindex.htm>

²³ *Scrabble* refers to the SCRABBLE® brand word game, a registered trademark of Milton Bradley, a division of Hasbro, Inc.

²⁴ A word of Yiddish origin defined as "expert" with a connotation of "know-it-all"

²⁵ Quackle is available at <http://web.mit.edu/jasonkb/www/quackle/>

- 26 ... and perhaps derives its name from this property.
- 27 "FISHING—Generally it's a bad idea to play 1 or 2 letters in the hope of picking up a specific letter to make a great play. The most common letter in Scrabble is E; there are 12 E's in the set out of a total of 100. This means on average, you only have a 1 in 8 chance of picking an E and that the chances of picking any other specific letters are lower. Near the end of the game, however, fishing becomes more viable. Obviously, you won't know precisely what letters your opponent holds. But you can see what letters have been played and take an educated guess at what's left in the bag."—<http://www.mattelscrabble.com/en/adults/tips/tip5.html>
- 28 <http://www.cs.ualberta.ca/events/csdays/1999/openhouse/chinook.html>
- 29 Also called Pachisi, it was popular in the Mughal courts of India. <http://www.tradgames.org.uk/games/Pachisi.htm>
- 30 Including himself, his brothers and his wife.
- 31 Rules of Backgammon © 1996-2007 by Tom Keith, <http://www.bkgm.com/rules.html>
- 32 The problem is circumvented in bridge to a large extent because performance is evaluated against other players on the same deal.
- 33 Move the checker at 13 to position 9, and the one at 6 to 5.
- 34 By the author's nephews, Anish (age 9) and Arnav (age 7), who invented the new rule.

Constraint Satisfaction Problems

Chapter 9

In state space search, we formulate problem solving as a process of making a sequence of moves from a given *Start* state ending in a desired *Goal* state. We assume that the moves themselves are generated by a move generating function *moveGen*. But states have to be represented in some language, and the *moveGen* function operates upon that representation. Often it makes sense to focus directly on those representations rather than view the problems at the abstract level of state space search. However, we have to be careful that our formulations are general enough to allow for the design of domain independent algorithms. In Chapter 7, we looked at the formulation of planning problems in which the *moveGen* function was replaced by its constituent operators.

In this chapter, we look at an alternate formulation of problems in which we describe the *constraints* that the *solution must satisfy*. The *Constraint Satisfaction Problem (CSP)* is a problem that specifies these constraints. The CSP is described as a set of variables, a set of domains for the variables where each domain contains allowable values for the variable, and a set of constraints that must be satisfied in any solution. A CSP, or a *constraint network*, is a triple (X, D, C) in which,

- X is a finite set of variables $\{x_1, x_2, \dots, x_n\}$,
- D is a set of domains $\{D_1, D_2, \dots, D_n\}$ with each domain D_i containing values that the corresponding variables can take, and
- C is a set of constraints $\{C_1, C_2, \dots, C_k\}$ on some subsets of X .

Each C_i is a relation R_i on a subset $S_i \subseteq X$ of a variable called the *scope* of C_i . The constraint C_i can be viewed as a pair $\langle S_i, R_i \rangle$. The relation R_i may be expressed as an intension, for example $x_j < x_k$, or if the domains are finite, it could be expressed as an extension. Without any loss of generality, we assume that there is at most one constraint for a given scope. If there are more than one constraints, one can always combine them by the logical *AND* operation, which in the extension form will manifest as set intersection.

The solution of a CSP is an assignment $\{\langle x_i, v_{ik} \rangle \mid x_i \in X, v_{ik} \in D_i\}$ for all variables of X such that every constraint in C is satisfied.

Observe that while the solution constitutes of a consistent assignment

of values of *all* variables, the constraints themselves are often local, each being over a few variables. If there are n variables in the CSP and the domain of each has k values, then a brute force algorithm would have to look k^n combinations in the worst case. We would like to do better than that, and as we shall see in the chapter, algorithms that work locally on a smaller set of variables can be employed to reduce the search space. The task of solving CSPs is approached with a two pronged strategy. The first is to prune the search space by a process of constraint propagation. The second is to search in an efficient manner. We look at both these strategies in this chapter.

The following is an example of a CSP with finite domains,

Problem CSP1

- $X = \{x_1, x_2, x_3\}$
- $D = \{D_1, D_2, D_3\}$ where $D_1 = D_2 = D_3 = \{1, 2, 3\}$
- $C = \{C_{12}, C_{23}\}$ where $S_{12} = \{x_1, x_2\}$ and $S_{23} = \{x_2, x_3\}$ and R_{12} and R_{23} are defined below.

The constraint C_{12} has scope $S_{12} = \{x_1, x_2\}$. We will use a shortcut to represent this fact. That is, the indices of S_{ij} will implicitly identify the variables x_{ij} and x_j as the variables that form the scope. The corresponding relation $R_{12} \subseteq D_1 \times D_2$ is a relation on the variables x_1 and x_2 . We use the notation $\langle x, y \rangle$ to stand for the pair of allowable values, $x \in D_1$ and $y \in D_2$ for the two corresponding variables, in this case x_1 and x_2 , and likewise for the other constraint C_{23} . We may specify the relations as an intension,

$$R_{12} = \{\langle x, y \rangle \mid x \in D_1, y \in D_2, \text{ and } x < y\}$$

and

$$R_{23} = \{\langle x, y \rangle \mid x \in D_2, y \in D_3, \text{ and } x < y\}$$

or we could specify them as extensions,

$$R_{12} = \{\langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 2, 3 \rangle\}$$

and

$$R_{23} = \{\langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 2, 3 \rangle\}$$

Observe that if the domains of the variables had been infinite, we could have still used the intensional form of the relation, but not the extensional form. There has been the development of a considerable number of techniques for solving constraint satisfaction problems dealing with numbers in specialized domains of mathematics, for example solving sets of linear equations. We will, however, focus on the general approach to solving CSP problems with *finite domains* in which the relations are expressed as extensions.

A CSP can be depicted by a constraint graph. Each node in a constraint graph represents a variable, and an edge connects two

variables if they participate in a constraint. The absence of an edge between two nodes means that all combinations of their values are allowed. One must be careful to understand that this means that *locally*, there is no constraint between the two variables, or that there is no *explicit* constraint between the two variables. Globally, there might only be certain combinations of values that participate in solutions. Figure 9.1 depicts the constraint graph for the problem CSP1. One can observe that there is an *implicit* constraint that $x_1 < (x_3 - 1)$ which does not find a place in the constraint graph because it is not explicit. But the relation will hold in any solution of the CSP.

We shall further restrict our focus on methods to solve binary CSPs. Binary CSPs are those CSPs where the constraints have scopes of sizes 1 or 2 only. For a binary CSP, the edges in the constraint graph in fact represent constraints. It has been observed that any nonbinary CSP can be converted to a binary CSP (see for example (Rossi et al., 1990), (Tsang, 1993), and (Mamoulis and Stergiou, 2001)). One way to do this is by introducing extra variables. Consider a variation of the above CSP containing a relation over all three variables.

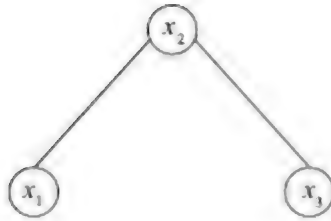


FIGURE 9.1 The constraint graph for the CSP1.

Problem CSP2

- $X = \{x_1, x_2, x_3\}$
- $D = \{D_1, D_2, D_3\}$ where $D_1 = D_2 = D_3 = \{1, 2, 3\}$
- $C = \{C_{123}\}$ where $S_{123} = \{x_1, x_2, x_3\}$ and

$$R_{123} = \{ \langle x, y, z \rangle \mid x \in D_1, y \in D_2, z \in D_3, \text{ and } x \neq y, y \neq z, x \neq z \}$$

As an extension,

$$R_{123} = \{ \langle 1, 2, 3 \rangle, \langle 1, 3, 2 \rangle, \langle 2, 1, 3 \rangle, \langle 2, 3, 1 \rangle, \langle 3, 1, 2 \rangle, \langle 3, 2, 1 \rangle \}$$

We can now introduce a new variable Z with the domain R_{123} and express the same CSP using binary relations as follows:

- $X = \{x_1, x_2, x_3, Z\}$
- $D = \{D_1, D_2, D_3, D_Z\}$ where $D_1 = D_2 = D_3 = \{1, 2, 3\}$ and $D_Z = R_{123}$
- $C = \{C_{1Z}, C_{2Z}, C_{3Z}\}$ where $S_{1Z} = \{x_1, Z\}$, $S_{2Z} = \{x_2, Z\}$, $S_{3Z} = \{x_3, Z\}$ and $R_{1Z} = \{ \langle 1, \langle 1, 2, 3 \rangle \rangle, \langle 1, \langle 1, 3, 2 \rangle \rangle, \langle 2, \langle 2, 1, 3 \rangle \rangle, \langle 2, \langle 2, 3, 1 \rangle \rangle, \langle 3, \langle 3, 1, 2 \rangle \rangle, \langle 3, \langle 3, 2, 1 \rangle \rangle \}$

The relation R_{1Z} contains pairs in which every value of x_1 is paired to those tuples of Z where the value of x_1 is the same. A similar definition can be given for the other two relations R_{2Z} and R_{3Z} . One can also express these relations using the projection operator of relational algebra in which the projections of the tuples in Z match the corresponding variable. The conversion to the binary CSP is done by adding extra variables. This is often needed because it has been shown that the number of binary CSPs that can be constructed with N variables is much smaller than the number of relations of N variables (Montanari, 1974).

In any case, a vast array of problems can be expressed as finite binary CSPs. Given the uniform manner in which all these problems are formulated, one can explore and exploit efficient techniques for solving them.

9.1 N-Queens

Consider the problem of placing N queens on an $N \times N$ chessboard, so that no queen attacks another. As a state space search problem, we would have modelled it as a problem of placing queens on an empty chessboard. The initial state of the problem is illustrated for $N = 6$ in Figure 9.2.

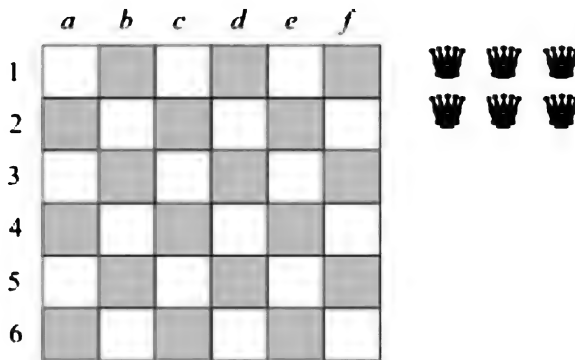


FIGURE 9.2 The six queen problem is to place the six queens on a 6×6 chessboard such that no queen attacks another.

One possible move in the state space formulation could be to *place* a queen on an empty square. The task is then to place the N queens on the board. As one can imagine, this would needlessly generate a huge search space since the first queen can be placed in N^2 ways, the second one in $(N^2 - 1)$ ways, and so on. An alternative state space formulation recognizes the constraint that no two queens can be on the same row or the same column. This allows us to describe the state as a vector of N values representing columns, with the index being identified with a queen. For example, the vector $\langle a, c, f, b, e, d \rangle$ represents a state in which the first queen is on column a , the second one on column c , and so on. It is

implicit in the representation that the first queen is on row 1, the second on row 2, and so on. As described in Chapter 3, one can search in the solution space by exploring permutations of the above vector.

The N -queens problem is a natural binary CSP because the constraints are on the placement of each pair of queens. We can formulate it as a set variables $\{Q_1, \dots, Q_N\}$ with the domains as the column identifiers. In our example, the columns are labelled with the letters of the English alphabet, as is the norm in chess literature. See Exercise 2 for an alternate notation that allows us to express the constraints in intensional form. The CSP for a 4-queen problem is given below (the 6-queen problem is left as an exercise),

Problem CSP3

- $X = \{Q_1, Q_2, Q_3, Q_4\}$
- $D = \{D_1, D_2, D_3, D_4\}$ where $D_1 = D_2 = D_3 = D_4 = \{a, b, c, d\}$
- $C = \{C_{jk} \mid 1 \leq j \leq k \leq 4\}$ where $S_{jk} = \{Q_j, Q_k\}$ and

with,

$$\begin{aligned} R_{12} = R_{23} = R_{34} &= \{ \langle a, c \rangle, \langle a, d \rangle, \langle b, d \rangle, \langle c, a \rangle, \langle d, a \rangle, \langle d, b \rangle \} \\ R_{13} = R_{24} &= \{ \langle a, b \rangle, \langle a, d \rangle, \langle b, a \rangle, \langle b, c \rangle, \langle c, b \rangle, \langle c, d \rangle, \langle d, a \rangle, \langle d, c \rangle \} \\ R_{14} &= \{ \langle a, b \rangle, \langle a, c \rangle, \langle b, a \rangle, \langle b, c \rangle, \langle b, d \rangle, \langle c, a \rangle, \langle c, b \rangle, \langle c, d \rangle, \langle d, b \rangle, \langle d, c \rangle \} \end{aligned}$$

The most straightforward approach to solve CSPs is by search, like in state space search. The difference is that in state space search algorithms studied in the preceding chapters, one can only test a board with all the queens placed, while in CSP, one can check the constraints for a partial placement as well. For example, after placing the first queen, one will try the second one only on locations that are allowed by the constraints between the first two queens. Further, it is possible to *look ahead* at the effects of partial placements to try and spot a problem well before it occurs. One can reduce the domains of all queens as and when each queen is placed. Assuming that we place queens in their given natural order, and place each queen on the leftmost available column, the situation after placing queen 1, and after the first four queens is shown in Figure 9.3 for a 6-queen problem. After the first queen is placed, the crosses on the board on the left indicate squares on which no queen can be placed. This is equivalent to deleting values that will not be allowed from the domains of each future queen¹.

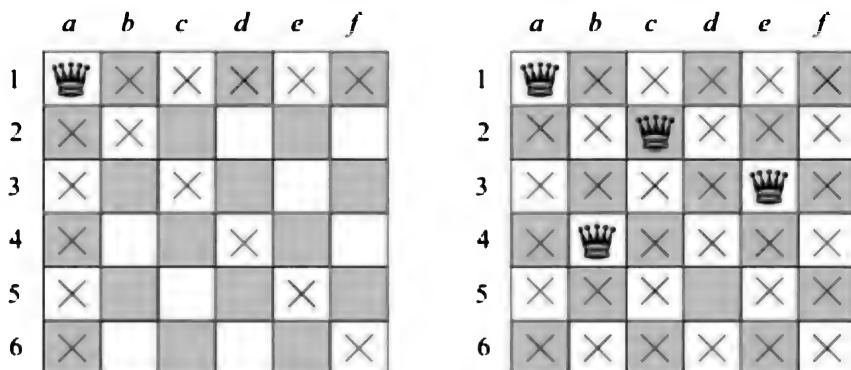


FIGURE 9.3 Placing each queen on the leftmost *available* column. The board after placing 1 queen is on the left and the after 4 queens is on the right. The crosses represent values deleted from the domains of all queens. We can see that after placing queen 4, there is no place left for queen 6.

As one can see from the figure on the right, after the first four queens have been placed, the domain of queen 6 has already become empty. Thus, a search algorithm can backtrack without even trying to place the fifth queen.

Before we look at search, we look at the idea of *constraint propagation* or *consistency establishment*. Even before starting search, one can do a certain amount of reasoning on the problem. Consider the problem *CSP1* described above. The relation R_{12} asserts that the value of the variable x_1 should be strictly less than the value of x_2 . This means that x_2 can never take the value $x_2 = 1$. We can thus remove the value 1 from the domain of x_2 so that a search algorithm will not even try it. We look at his notion of constraint propagation in the next section.

9.2 Constraint Propagation

A solution to a *CSP* is an assignment of a value for each variable from its domain, such that all constraints are satisfied. A search algorithm tries different assignment of values to variables in some order. The domains of variables contain all possible values that each variable can take. Some of them may be part of a solution and some may not. The algorithm *Backtracking* described in a Section 9.6 below tries only those values for the next variable that are consistent with the choices made for earlier variables. If we can reduce the values of the domain in a variable, we can reduce the number of options a search algorithm needs to look at. In the extreme case, we can make the search *backtrack free* wherein all consistent values available for each variable at its turn lead to a solution. Reduction of domains is a form of making inferences, and like all computation incurs a cost. The goal is to arrive at a balance in which the cost of making inferences is less than the reduction in the cost of search.

We introduce the notion of *consistency* for a *CSP*. There are various

degrees of consistency defined as follows. Given that $(k - 1)$ variables have been assigned values, a CSP is said to be k -consistent, if for any k^{th} variable there exists a value that is consistent with the $(k - 1)$ variables. We need a few more definitions to formalize this notion (Dechter, 2003).

An *instantiation* or a *compound label* of a subset of variables V is the assignment of values to each variable from its domain. If the set of variables is $V = \{x_{V1}, x_{V2}, \dots, x_{Vk}\}$ then the instantiation is the tuple of ordered pairs $\langle x_{V1}, a_{V1} \rangle, \langle x_{V2}, a_{V2} \rangle, \dots, \langle x_{Vk}, a_{Vk} \rangle$ where each pair consists a variable and a value or label assigned to it. We also write the instantiation as $\langle x_{V1} = a_{V1}, x_{V2} = a_{V2}, \dots, x_{Vk} = a_{Vk} \rangle$ or as a vector $\bar{a} = \langle a_{V1}, a_{V2}, \dots, a_{Vk} \rangle$.

An instantiation \bar{a} *satisfies* a constraint $\langle S, R \rangle$ iff it is defined over the variables in S and the components of \bar{a} corresponding to S are present in R . We also say that the instantiation is k -satisfiable. For example, the instantiation $\langle b, d \rangle$ for $\{Q_1, Q_2\}$ satisfies R_{12} . An instantiation \bar{a} over the set of variables V is *consistent* if it satisfies all the constraints in the CSP whose scopes are subsets of V . We say that the instantiation is k -consistent if it has k values. For example, the assignment $\langle a, b, c, e \rangle$ for $\{Q_1, Q_2, Q_3, Q_4\}$ is consistent because it satisfies $R_{12}, R_{13}, R_{14}, R_{23}, R_{24}$ and R_{34} .

A *solution* of a CSP is a consistent assignment over all its variables. The CSP is said to *express* the set of solutions of the CSP. Two CSPs (X, D, C) and (X, D', C') are said to be *equivalent* if they express the same set of solutions. This is of particular interest when D' contains reduced domains arrived at by constraint propagation. Search on D' would be less expensive and yield the same solutions.

9.2.1 Node Consistency

A CSP is said to be *node consistent* or *1-consistent* if and only if every value in every domain satisfies the unary constraints on the corresponding variable. A unary constraint, by definition, selects a subset of a domain. For example, if the domain $D_x = \{1, 2, 3, 4, 5\}$ and there is a constraint $C_x = (\{x\}, \{x < 4\})$ in a CSP then for that CSP the values 4 and 5 can be removed from D_x because they can never be part of a solution. The algorithm for achieving node consistency inspects all values in all domains to check that they are included in the corresponding constraint. It removes values from the domain of a variable if that value is not found in the extension of the relation for the corresponding unary constraint. One must remember that the absence of an explicit constraint is equivalent to a universal constraint that says that all values are allowed. The algorithm *NodeConsistency* is given in Figure 9.4.

```

NodeConsistency( $X, D, C$ )
1  for each  $x \in X$ 
2    for each  $v \in D_x$ 
3      if  $v \notin R_x$  where  $C_x = (\{x\}, R_x)$ 
4        then remove  $v$  from  $D_x$ 
5  return ( $X, D, C$ )

```

FIGURE 9.4 The algorithm *NodeConsistency* takes as input a CSP and returns an equivalent node consistent CSP.

9.2.2 Arc Consistency

Arc Consistency is concerned with finding consistent values for pairs of variables. In the constraint graph, every pair of variables x and y that participate in some constraint are linked by an edge (x, y) . For a binary CSP every edge represents a constraint R_{xy} . Let variable x have domain D_x and variable y have domain D_y . An edge (x, y) in the constraint graph is said to be *arc consistent* if for every value v_x in D_x , there exists a value v_y in D_y such that $\langle v_x, v_y \rangle \in R_{xy}$, and *vice versa* for v_y and v_x . We say that v_y supports v_x , and *vice versa*.

One can depict the relation R_{xy} with a *matching diagram* in which edges connect *values* from the domains D_x and D_y (Dechter, 2003). In an arc consistent edge (x, y) every value in each domain is connected to some value in the other domain in the matching diagram. Figure 9.5 shows an example of a matching diagram where the edge (x, y) is not arc consistent on the left. One must keep in mind that the edge in the constraint graph is (x, y) , while the edges in the matching diagram link *values* from the domains of the two variables.

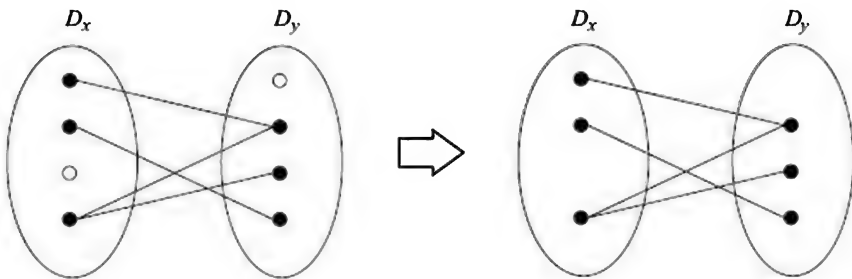


FIGURE 9.5 The matching diagram for a relation R_{xy} . On the left, the pair of variables (x, y) are not arc consistent. On the right, they become arc consistent after the non-participating values are removed from the domains.

The task of achieving arc consistency is to remove those values from each domain that are not the endpoints of any link, or that are not supported by any value of the other variable. The figure on the right shows the edge (x, y) , after arc consistency has been enforced. An edge (x, y) can be made arc consistent by inspecting the two domains, D_x and

D_y , and removing the unsupported values. If in the process, any domain becomes empty then it means that the CSP is *inconsistent* and does not have a solution. The procedure *Revise* given in Figure 9.6 reduces the domain D_x with respect to the edge (x, y) (Mackworth, 1977). Observe that this will have to be called twice, once with each variable, to enforce that the edge is arc consistent. If there are k values in each domain, the complexity of this algorithm is $O(k^2)$. Not shown in the algorithm, but in practice if the *if* condition is found true in line 4 then one would exit the inner for loop after line 5, thus saving on some computation.

```

Revise( $D_x, D_y, R_{xy}$ )
1  for each  $v \in D_x$ 
2    delete  $\leftarrow$  true
3    for each  $w \in D_y$ 
4      if  $(v, w) \in R_{xy}$ 
5        then delete  $\leftarrow$  false
6    if delete = true
7      then remove  $v$  from  $D_x$ 
8  return  $D_x$ 

```

FIGURE 9.6 The algorithm *Revise* takes as input the domains of two variables forming an edge and reduces the first domain, so that every value in D_x has a corresponding value in D_y . In practice, when a matching value is found in D_y the algorithm will exit that loop.

A CSP is said to be arc consistent if every edge in its constraint graph is arc consistent.

At first thought, it might appear that making a CSP arc consistent is simply a matter of looking at the domain of each variable with respect to every edge. However, deleting a value from a domain may have a cascading effect because that value might have been the supporting value for a value in another domain. Figure 9.7 extends the CSP of Figure 9.5 with two more variables, z and w . To achieve arc consistency on the four variables, *Revise* has to be called more than once for some pairs of variables. The matching diagram on the top depicts the original problem, and the diagram below it is the state after one round of calls to algorithm *Revise* left to right. As one can see, two values in D_y which had matching values in D_z after (y, z) was made arc consistent, lost their supports after (z, w) was made arc consistent.

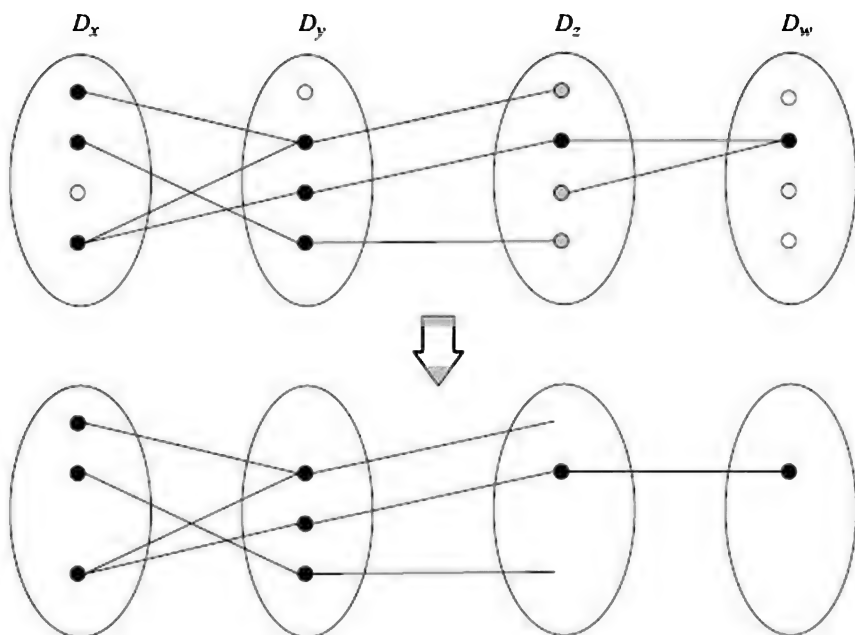


FIGURE 9.7 The domains of four variables x , y , z and w are shown on the top along with the relations R_{xy} , R_{yz} , and R_{zw} . After one round of arc consistency with (x, y) , (y, x) , (y, z) , (z, y) , (z, w) and (w, z) , the matching diagram is shown below. As one can see, two values in D_y are unsupported at this stage.

The reader is encouraged to continue applying the *Revise* algorithm to the four variables until the three edges in the constraint graph, (x, y) , (y, z) and (z, w) are arc consistent.

We shall look at three well known algorithms to achieve arc consistency (Mackworth, 1977). The simplest algorithm AC-1 to achieve arc consistency repeats the complete cycle of calls to *Revise*, until no domain changes in a cycle. It is computationally the most expensive of the three. The algorithm is given in Figure 9.8.

```

AC-1 ( $X, D, C$ )
1  repeat
2    for each  $x \in X$ 
3      for each  $y \in X$  such that  $C_{xy} \in C$ 
4         $D_x \leftarrow \text{Revise}(D_x, D_y, R_{xy})$ 
5         $D_y \leftarrow \text{Revise}(D_y, D_x, R_{xy})$ 
6  until no domain is changed
7  return ( $X, D, C$ )

```

FIGURE 9.8 The algorithm AC-1 takes as input a CSP and returns an equivalent CSP that is arc consistent. It calls algorithm *Revise* with all edges in the constraint graph till quiescence is reached.

Let there be e constraints in the CSP. Given e edges in the constraint

graph and at most k values per domain then one cycle of AC-1 has complexity $O(ek^2)$. If there are n variables then the maximum number of values is nk , and in the worst case one value is removed per cycle. Therefore, in the worst case, the complexity of AC-1 is $O(nek^3)$. If only a few variables participate in constraints then there can be at most $2e$ variables, two for each constraint. The complexity can then be estimated to be $O(e^2k^3)$.

The algorithm AC-1 blindly applies Revise to all pairs of variables, irrespective of whether their domains have changed or not. The algorithm AC-3² keeps tracks of removal of elements from domains. Only if a value for variable x is removed, does AC-3 look again at other variables connected to x via some edge.

```

AC-3 (X, D, C)
1 queue ← ( )
2 for each pair (xk, xj) that participates in a constraint Ckj
3   enqueue ((xk, xj), queue)
4   enqueue ((xj, xk), queue)
5 while not(empty(queue))
6   varPair ← dequeue(queue)
7   x ← Head(varPair)
8   y ← Head(Tail(varPair))
9   Dx ← Revise(Dx, Dy, Rxy)
10  if Dx has changed
11    then for each z which is connected to x such that z ≠ x and z ≠ y
12      enqueue ((z, x), queue)
13 return (X, D, C)

```

FIGURE 9.9 Like AC-1, the algorithm AC-3 first calls *Revise* for each pair of variables that participate in a constraint. After the first round however, AC-3 looks at only those domains whose values might have lost support because of the removal of some elements by *Revise*.

Algorithm AC-3 processes each constraint at most $2k$ times, where k is the size of the domain of each variable. This is because it looks at a pair (or constraint) again, *only* if a value has been removed from one of the domains at its two ends. Since there are e binary constraints, the complexity of AC-3 is $O(ek^3)$. The factor k^2 as before comes from the complexity of *Revise*.

While AC-3 is more efficient than AC-1, it turns out that it is still not optimal. This is because when it relooks at a pair of variables, it inspects their entire domain. One can do better by keeping track of which values may have lost their support and only processing those values (Mohr and Henderson, 1986). In order to do this, however, additional data structures have to be maintained. The algorithm operates at the level of labels. Each label $\langle x, a \rangle$ must have a supporting value for each constraint that the variable x participates in. The following data structures are used.

- Support set S . S is set of sets named $S_{\langle x, a \rangle}$, one for each variable value pair $\langle x, a \rangle$. For each variable-value pair (or label) $\langle x, a \rangle$, the support set contains a list of supporting labels.

$$S_{\langle x, a \rangle} = \{ \langle y, b \rangle \mid y \in X, b \in D_y, \text{ and } \langle a, b \rangle \in R_{xy} \}$$

The support set S can be constructed by inspecting the pairs of variables for each constraint. Given e constraints and domain sizes k , this step is $O(ek^2)$.

- Counter array *counter*. For each label $\langle x, a \rangle$, the counter array maintains the number of supports from a variable y . If this value becomes zero, it means that the value 'a' has to be removed from D_x .

$$\text{counter}(x, a, y) = \text{number of values in } D_y \text{ that support } \langle x, a \rangle$$

The counter array can be constructed along with S , adding a constant amount of computation for each label.

- A queue *queue* of labels without support that need to be processed.

The resulting algorithm AC-4 is given in Figure 9.10. We separate the initial setting up of the data structures and the propagation into two modules. Before propagation begins, the procedure *Initialize* constructs the data structures. For every constraint C_{xy} it does the following. For each label $\langle x, a \rangle$ in D_x it finds the supports $\langle y, b \rangle$ in D_y and stores them in $S_{\langle x, a \rangle}$, and conversely stores $\langle x, a \rangle$ in $S_{\langle y, b \rangle}$ for each label $\langle y, b \rangle$ in D_y . It also counts how many support each label has for each constraint it participates in. Finally, it identifies unsupported labels, removes them from their domains, and enters them into a queue to propagate the effects of their being removed.

In the propagation phase after the initialization, it removes the labels in the queue and deletes them one by one. For every value a for a variable x that is removed, the AC-4 algorithm looks up $S_{\langle x, a \rangle}$ for values of other variables it was supporting. For each such value b of variable y , it decrements $\text{counter}(y, b, x)$. If any counter becomes zero then the corresponding value is removed and a label is added to *queue* for propagation.

Thus while AC-3 was an improvement over AC-1 because it selectively calls *Revise* only with those pairs of domains for which a supporting a value has been deleted, AC-4 is an improvement over AC-3 because it only looks at those labels whose supporting labels have been deleted. Since it does not make a brute force call to revise it saves on the $O(k)^2$ computation that *Revise* does.

Given that there are e edges and each domain has at most k values, the complexity of procedure *Initialize* is $O(ek^2)$. This is because constructing the support S and the counter can be done by inspecting the domains of the variables on each side of each constraint. In the algorithm above, the support sets have been initialized outside the loop at step 4 for simplicity. In practice, they will be constructed inside the loop and only for those labels that are parts of constraints. The queue of unsupported labels is also constructed inside the loop. The size of S is also $O(ek^2)$, which is also the value of the *sum of all counters*. Since in the worst case in the while loop of algorithm AC-4 one counter will get decremented in each cycle, there can be at most $O(ek^2)$ cycles, and therefore the

complexity of AC-4 is $O(ek^2)$. Given that the various data structures have to be stored for AC-4, it can be shown that its space complexity is also $O(ek^2)$.

```

AC-4 (X, D, C)
1 <queue, S, counter> ← Initialize(X, D, C)
2 while not(empty(queue))
3   varVal ← dequeue(queue)
4   x ← Head(varVal)
5   a ← Head(Tail(varVal))
6   for each label <y, b> in S<x, a>
7     counter(y, b, x) ← counter(y, b, x) - 1
8     if counter(y, b, x) = 0
9       then Dy ← Dy \ b
10      enqueue ((y, b), queue)
11 return (X, D, C)

Initialize (X, D, C)
1 queue ← ( )
2 for each label <x, a>
3   S<x, a> ← ( )
4 for each constraint Cxy = ({x, y}, Rxy)
5   for each b ∈ Dy
6     counter(y, b, x) ← 0
7     for each a ∈ Dx
8       counter(x, a, y) ← 0
9       for each b ∈ Dy
10        if <a, b> ∈ Rxy
11          counter(x, a, y) ← counter(x, a, y) + 1
12          counter(y, b, x) ← counter(y, b, x) + 1
13          S<y, b> ← S<y, b> ∪ {<x, a>}
14          S<x, a> ← S<x, a> ∪ {<y, b>}
15        if counter(x, a, y) = 0
16          Dx ← Dx \ a
17          enqueue ((x, a), queue)
18        if counter(y, b, x) = 0
19          Dy ← Dy \ b
20          enqueue ((y, b), queue)
21 return <queue, S, counter>

```

FIGURE 9.10 Unlike AC-1 and AC-3, AC-4 operates at the level of labels, or <variable value> pairs. For this, it has to do elaborate bookkeeping, keeping track of individual edges in the matching diagram, and degree of each vertex looking out for vertices that get isolated.

The complexity measures described above are worst case complexities. They are also asymptotic complexities in which the role of constants does not show up in the comparison. It has been suggested that for some problems, AC-3 might in fact be a better algorithm on the average than AC-4 (Van Hentenryck, 1989). Furthermore, the performance of AC-3 is critically dependent on the order in which the variables are processed (Wallace and Freuder, 1992). It has been argued, with support from empirical evidence, that on the average AC-3 yields better performance than AC-4 for most problems (Wallace, 1993).

9.3 Scene Labelling

One interesting problem where constraint propagation has been applied with remarkable effect is the scene labelling problem, also known as the Huffman-Clowes scene labelling problem (Clowes, 1971), (Huffman, 1971).

The scene labelling problem is a classification problem in which the lines in an image have to be assigned a label describing them. Each edge in the diagram needs to be assigned a label from the set $\{+, -, \rightarrow, \leftarrow\}$. Consider the following vertex that is part of a line drawing of a scene.

Now given that there are four possible labels for each of the three edges, the edges meeting at the vertex can be labelled in $4^3 = 64$ different ways. However, if the line diagram is a scene depicting physical objects then the labels of the three edges are constrained. On the one hand, they are constrained by what combinations of labels at vertices are possible for physical objects. On the other hand, the labels will have to be consistent with the vertices that the edges are connected to at the other end, because only one label can be applied for one edge. Let us assume that the line drawings are restricted to trihedral objects with plane surfaces in which exactly three edges, or planes, meet at a vertex.

The meanings of the labels are as follows,

- $+$: The edge is convex when the two faces enclose material within an angle less than 180° .
- $-$: The edge is concave when the two faces enclose material within an angle greater than 180° .
- \rightarrow : The edge is a boundary edge with visible face below the edge.
- \leftarrow : The edge is a boundary edge with visible face above the edge.

Observe that the two boundary edges essentially say that the material is on the right-hand side as you traverse along the arrow on the edge, but since either direction can be a label, we have treated them as two distinct labels. Figure 9.12 shows examples of two objects within the scope of this labelling with their labels, and three line drawings that are not of trihedral objects without labels.



FIGURE 9.11 A vertex with three edges.

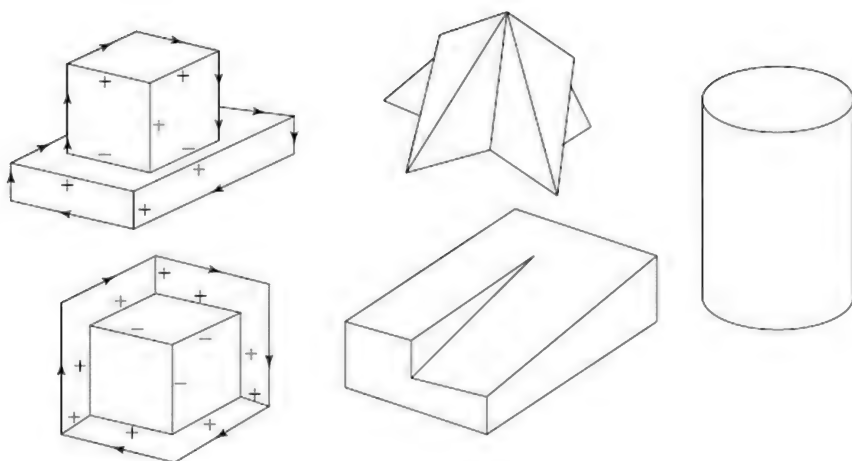


FIGURE 9.12 Trihedral objects with their labels on the left, and nontriheral objects on the right. Observe that the vertex of Figure 9.11 occurs in both with different labels. The two figures in the middle have a vertex with less than or more than three edges, and the cylinder has a curved surface. The bottom figure in the middle too has a surface that is not planar.

One can identify four kinds of vertices as seen in the line drawings. They are known as the *W*, *Y*, *T* and *L* vertices. In the *L* type of vertex, only two edges are visible, while the others have three edges visible. Figure 9.13 depicts the four kinds of vertices and their representation as lists of edges. Given that an edge can get one of four labels, there are 64 different labels for three edge vertices, and 16 for the *L* vertex. It turns out that only a small number of label combinations are valid for objects in which vertices are made of exactly three planar faces. We assume that the objects are viewed such that vertices are not formed because of a particular viewpoint. For example, the two collinear line segments of a *T* vertex are part of the same edge, and not two different edges appearing collinear because of the viewpoint. The twenty valid combinations are also shown in the figure. Observe that three label combinations of the *Y* joint are rotations of each other. This is because the edge labelled “-” could have been in any orientation.

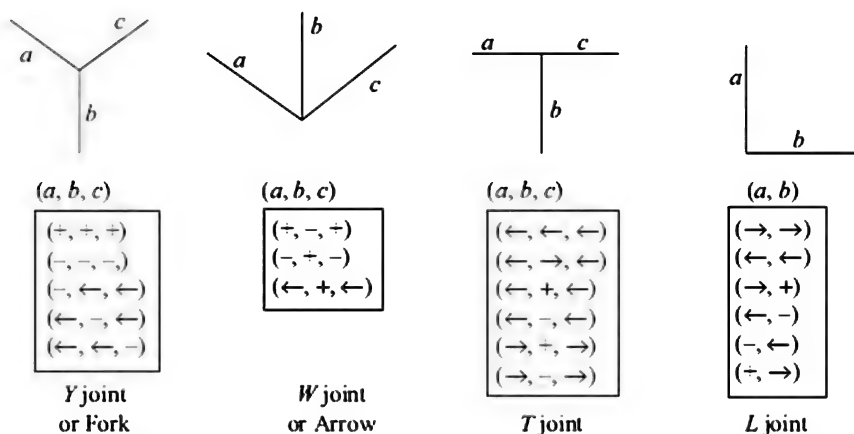


FIGURE 9.13 The four types of vertices and their list notation. The allowable combinations of labels are given below in boxes. The edges can be identified unambiguously irrespective of their orientation except for the *Y vertex*.

The allowable labels for edges meeting at vertices are determined by the physical properties of the sort of objects we are dealing with. These are the possible label combinations when the vertices are seen in isolation. When vertices correspond to physical objects, each edge of a vertex connects it to another vertex. Since the edge is the same for both vertices, it should be classified with the same label, and this imposes further constraints on the allowed labelling. This inter-vertex constraint is illustrated in Figure 9.14. Vertices V_1 and V_2 share an edge, as do V_2 and V_3 . The matching diagrams for these three vertices are also shown. The reader must remember that the labels (x, y, z) are with respect to the orientations shown in Figure 9.13. Only two labels of V_2 have support from V_1 and two from V_3 , and only one $(+, +, +)$ has support from both. Enforcing arc consistency will keep only this label for V_2 and the matching labels for V_1 and V_3 .

A CSP for the edge labelling problem can be posed as follows. Each vertex in the line diagram is a variable. The domains of the vertices are the (compound) labels as shown in Figure 9.13. Each edge in the diagram imposes a binary constraint on the two vertices it connects. The constraint is that the edge must be labelled *identically* from the set $\{+, -, \rightarrow, \leftarrow\}$ for the two vertices.

The CSP formulation for a simple object is shown in Figure 9.15. The object is shown on the upper left. The six vertices are $V_1 = (a, b, c)$, $V_2 = (c, d)$, $V_3 = (e, a)$, $V_4 = (b, f, g)$, $V_5 = (h, f, e)$, and $V_6 = (d, g, h)$. Each edge is represented as a constraint between two variables. The constraint is that a particular component of one label must match a particular component of the other label. We use notation $V_k(m) = V_j(n)$ to say that the m^{th} label of V_k must match the n^{th} label of V_j . For example, we have $V_6(2) = V_4(3)$, which says that the common edge “ g ” must get matching labels from both ends, V_4 and V_6 .

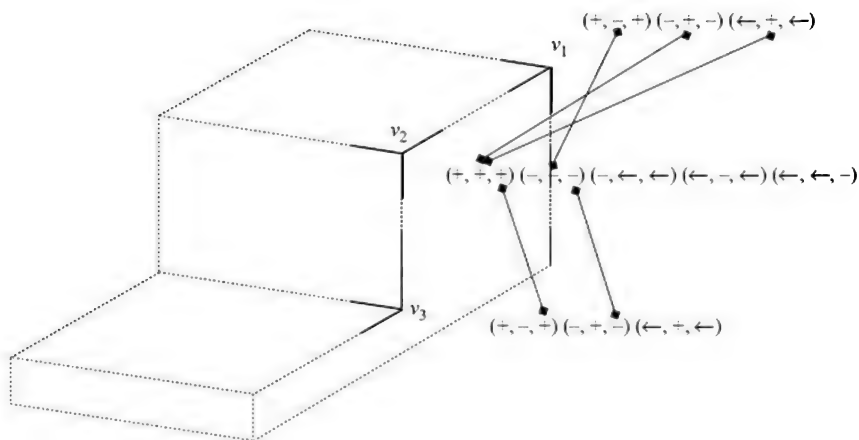


FIGURE 9.14 Constraints across vertices. Every edge connects two vertices and imposes a constraint that the labels on the two vertices must assign the same value for that edge. The matching diagram for three vertices V_1 , V_2 and V_3 is shown. Achieving arc consistency will remove unsupported labels.

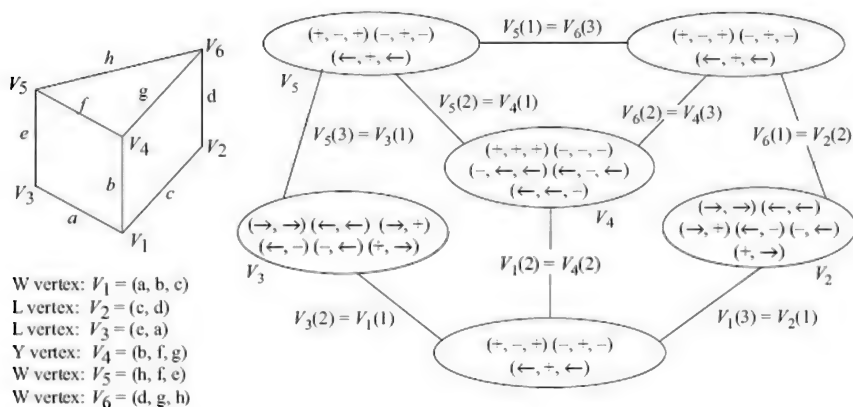


FIGURE 9.15 A simple object and its constraint network.

The reader is encouraged to enforce arc consistency on the above CSP.

When we consider a larger variety of objects and also allow shadows from illumination and cracks, the number and types of vertices increases dramatically. David Waltz considered these objects and introduced several additional types of junctions with four or more incident edges. Allowing for additional labels to represent shadows, cracks, overlap of objects, the number of physically realizable vertex labels reduces to 1790 from several billion combinations (Winston, 1992). The algorithm developed by Waltz for propagating constraints is essentially an arc-consistency algorithm similar to AC-3 algorithm described above (Waltz, 1975). The experiments conducted by Waltz showed that the constraint propagation pruned the domains of the edges dramatically, often to only

the labels occurring in a solution.

9.4 Higher Order Consistency

Arc consistency or 2-consistency ensures that only matching labels from two related domains remain in a CSP, and for every label in a domain, there is a supporting label in a neighbouring domain. But achieving arc consistency does not mean that the variables that remain will necessary be part of a solution. In fact, it is even possible that there might not be a solution. Consider the map colouring problem with three nodes and two colours.

Problem CSP4

$$(X, D, C) = (\{x_1, x_2, x_3\}, \{\{r, b\}, \{r, b\}, r, b\}, \{x_1 \neq x_2, x_1 \neq x_3, x_2 \neq x_3\})$$

This problem is already arc consistent, but there is no solution to the problem. This problem is not *path consistent*.

9.4.1 Path Consistency

A CSP is said to be *path consistent* or *3-consistent* if every pair of consistent variable instantiations can be extended to a third variable. Let the given instantiation or compound label be $\langle x = a, y = b \rangle$. This instantiation is consistent if $\langle a, b \rangle \in R_{xy}$. The two variable set $\langle x, y \rangle$ is said to be path consistent with respect to a variable z if for every such compound label $\langle a, b \rangle$, there exists a value $c \in D_z$ such that $\langle a, c \rangle \in R_{xz}$ and $\langle c, b \rangle \in R_{zy}$. That is, there is a path from a to b via c . Observe that we do not require the variables to participate only in explicit relations because we can take the absence of an explicit constraint as representing a universal relation that allows any pairing of values. As we shall see, enforcement of path consistency results in tightening of all relations, removing pairs that are not path consistent. In the process, a new explicit relation may be inferred.

Arc consistency ensures that for every value in the domain of a variable, there is a supporting value in a related domain. It ensures that every node participates in an edge in the matching diagram. Path consistency ensures that for every edge between a pair of values, there is a value in every other domain such that the end points of the edge participate in edges to the new value. It thus ensures that every edge participates in a triangle. This implies that a partial solution made up of the two values at the end points of the edge can be extended to any other variable. Recall that the complete solution is made up by consistent values for all the variables. The procedure *Revise-3* given in Figure 9.16 below is analogous to the procedure *Revise*, except that it removes

edges from the binary relation that cannot be extended to a triangle.

```

Revise-3( $D_z, R_{xy}, R_{xz}, R_{yz}$ )
1 for each  $(a, b) \in R_{xy}$ 
2    $delete \leftarrow true$ 
3   for each  $c \in D_z$ 
4     if  $(a, c) \in R_{xz}$  and  $(c, b) \in R_{yz}$ 
5       then  $delete \leftarrow false$ 
6   if  $delete = true$ 
7     then remove  $(a, b)$  from  $R_{xy}$ 
8 return  $R_{xy}$ 

```

FIGURE 9.16 The algorithm *Revise-3* takes as input the binary relations between three variables x, y , and z and the domain D_z . It ensures that every pair (a, b) in R_{xy} has pairs in R_{xz} and R_{yz} . If not, it removes the pair (a, b) from R_{xy} .

The complexity of *Revise-3* is $O(k^3)$ because there are $O(k^2)$ edges in R_{xy} in the worst case, and for each edge, the entire domain D_z has to be inspected in the worst case.

Figure 9.17 shows what happens after a call to *Revise-3* $((x, y), z)$, given four variables w, x, y and z . We assume the $R_{xy}, R_{xz}, R_{xw}, R_{yz},$ and R_{yw} are defined explicitly, and that R_{wz} is a universal relation. One must keep in mind that these binary relations are all symmetric. The black nodes in the matching diagram are the ones that emerge as consistent triangles in *this* call. The dashed edges between D_x and D_y in the figure on the left are the ones removed by the call *Revise-3* $((x, y), z)$. Of the two surviving edges, between the domains of x and y , the dashed one in the figure on the right would get removed, if a subsequent call to *Revise-3* $((x, y), w)$ were to be made. The first two edges were deleted because they did not have a corresponding value in D_z and the third one because there was no matching value in D_w .

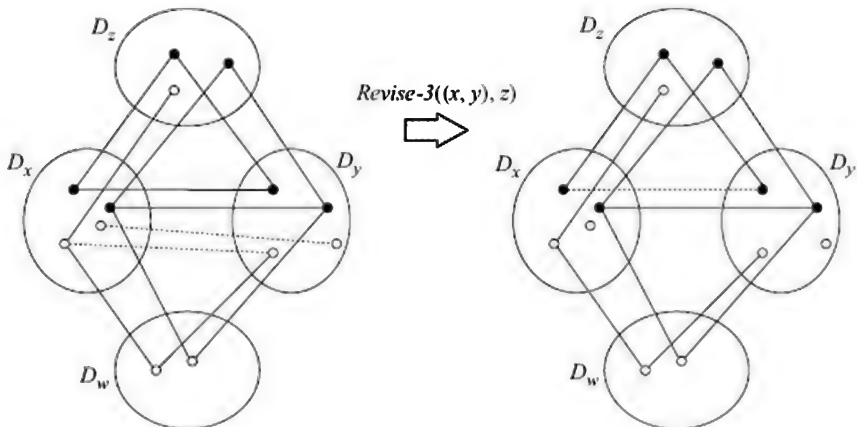


FIGURE 9.17 From the matching diagram on the left, a call to *Revise-3* $((x, y), z)$ removes the

two edges shown in dashed lines. Of the remaining two edges between D_x and D_y in the figure on the right, the dashed edge will get removed by a call to $Revise((x, y), w)$. Note that neither diagram is arc consistent.

The objective of path consistency is to arrive at a matching diagram of all variables, such that any edge between any two variables forms triangles with values in all other domains. The simple procedure to achieve path consistency PC-1 is analogous to the procedure AC-1. It makes calls to all possible combinations of variable pairs with all other variables, till no relation is pruned further. The algorithm is given in Figure 9.18.

```

PC-1 ( $X, D, C$ )
1 repeat
2   for each  $z \in X$ 
3     for each  $x \in X$  and each  $y \in X$  s.t.  $x \neq y \neq z$ 
4        $R_{xy} \leftarrow Revise-3(D_z, R_{xz}, R_{yz}, R_{xy})$ 
5   until no domain is changed
6 return ( $X, D, C$ )

```

FIGURE 9.18 The algorithm PC-1 takes as input a CSP and returns an equivalent CSP that is path consistent. It calls algorithm $Revise-3$ with all triplets in the constraint graph, till quiescence is reached. The output CSP contains pruned sets of relations.

The algorithm PC-1 looks at all pairs of variables, irrespective of whether there is an explicit constraint between them or not. This is because the absence of a constraint is equivalent to a universal constraint in which all combinations are allowed. Thus, in each cycle, the algorithm will inspect $(n - 1)^2$ edges for each of the n variables, expending $O(k^3)$ computations in each call to $Revise-3$. Thus, in each cycle, the algorithm will do $O(n^3k^3)$ computations. Further, in each cycle, in the worst case, we will remove one pair of values $\langle a, b \rangle$ from some constraint R_{xy} . In the worst case then, the number of cycles is $O(n^2k^2)$, because there are n^2 relations and each may have k^2 elements. Thus in the worst case, algorithm PC-1 will require $O(n^5k^5)$ computations.

Every time a call is made to $Revise-3$ $O(k^3)$, computations are needed. But can we cut down on the number of calls to $Revise-3$? The answer is indeed yes, and in a manner analogous to AC-3 the algorithm PC-2 only calls $Revise$ when a pair is removed from a relation. When $\langle a, b \rangle$ is removed from R_{xy} , one needs to check again whether $\langle a, b \rangle$ was a supporting edge for pairs $\langle c, a \rangle$ or $\langle c, b \rangle$, for all c . The algorithm PC-2 described below maintains a queue of triples which it needs to inspect in a manner similar to algorithm AC-3.

```

PC-2 (X, D, C)
1  queue ← ( )
2  for each  $x_i$ ,
3    for each  $x_k$  and  $x_j$  such that  $k < j$ ,  $k \neq i$ ,  $j \neq i$ 
4      enqueue ((( $x_k$ ,  $x_j$ ),  $x_i$ ), queue)
5  while not(empty(queue))
6    varTriple ← dequeue(queue)
7    x ← Head(Head(varTriple))
8    y ← Head(Head(Tail(varTriple)))
9    z ← Head(Tail(varTriple))
10    $R_{xy} \leftarrow \text{Revise-3}(D_x, R_{xy}, R_{xz}, R_{zy})$ 
11   if  $R_{xy}$  has changed
12   then for each z such that  $z \neq x$  and  $z \neq y$ 
13     enqueue (((z, x), y), queue)
14     enqueue (((z, y), x), queue)
15 return (X, D, C)

```

FIGURE 9.19 Like AC-3, the algorithm PC-3 maintains a queue of calls to *Revise-3* for each triple of variables. Subsequently, if it removes a pair from a relation, then it checks whether a side of a triangle with a third value is not being broken.

The complexity of PC-2 depends upon how many elements can be added to the queue. Every time an element is removed from the queue, the call to revise does a computation of $O(k^3)$. The minimum number of calls is $O(n^3)$, which is the number of distinct calls that can be made. In the worst case, in each cycle of the while loop, one pair is removed from one variable, two elements are added to the queue. For each combination $((x, y), z)$, at most k^2 elements can be removed from the relation, and therefore at most, $2k^2$ elements can be added to the queue. One can then argue that while loop executed at most is $O(n^3k^2)$ times, therefore the complexity of PC-2 is $O(n^3k^5)$.

Both PC-1 and PC-2 rely on calls to *Revise-3*, which inspects all values in the three domains. One can design a finer grained algorithm analogous to AC-4. Indeed such an algorithm was devised by Mohr and Henderson (1986) and has complexity $O(n^3k^3)$. Notice that like in the reduction in complexity from AC-3 to AC-4, here too complexity is reduced by a factor that was contributed to by the *Revise* operator. In particular, PC-4 looks at only the effect of removing a pair $\langle a, b \rangle$ rather than all the k^2 pairs that exist in the corresponding relation. We leave it as a complex exercise for the interested reader the design of the support structures needed.

It is worth noting that path consistency does not automatically imply arc consistency. The following example is due to Freuder (1982).

Problem CSP6

$$(X, D, C) = (\{x_1, x_2, x_3\}, \{\{r\}, \{r, b\}, \{r\}\}, \{\{\{x_1, x_2\}, x_1 \neq x_2\}, (\{x_2, x_3\}, x_2 \neq x_3)\})$$

As one can see, the CSP is path consistent, but it is not arc consistent. In general, if a CSP is i -consistent, it does not mean that it is $(i - 1)$ -consistent as well.

9.4.2 i -consistency

The algorithms for path consistency give us a clue of how to write algorithms for higher order consistency. The notion of i -consistency for a constraint network is that any consistent compound label for $i - 1$ variables can be extended by one more variable. The network is said to be strongly i -consistent if it is also j -consistent for all $j < i$. We can write a generalized consistency implementation algorithm by writing a generalized Revise-I algorithm as shown in Figure 9.20. For each variable z in the domain, the generalized i -consistency algorithm IC-1 looks at all subsets of variables of size $i - 1$ to test if *all* consistent instantiations of size $i - 1$ can be extended by a value from the domain D_z .

```

IC-1 ( $X, D, C$ )
1 repeat
2   for each  $z \in X$ 
3     for each  $S = \{x_1, x_2, \dots, x_{i-1}\} \subset X$  s.t. each  $x_i \neq z$ 
4        $R_S \leftarrow \text{Revise-I}(S = \{x_1, x_2, \dots, x_{i-1}\}, z)$ 
5   until no relation is changed
6 return ( $X, D, C$ )

Revise-I ( $S = \{x_1, x_2, \dots, x_{i-1}\}, x_i$ )
1 for each  $(a_1, a_2, \dots, a_{i-1}) \in R_S$ 
2   delete  $\leftarrow \text{true}$ 
3   for each  $c \in D_i$ 
4     if  $(a_1, a_2, \dots, a_{i-1}, c)$  is consistent
5       then delete  $\leftarrow \text{false}$ 
6   if delete = true
7     then remove  $(a_1, a_2, \dots, a_{i-1})$  from  $R_S$ 
8 return  $R_S$ 

```

FIGURE 9.20 The algorithm *Revise-I* takes as input a set of $(i - 1)$ variables and a distinct variable x_i , and identifies those labels that cannot be consistently extended. The generalized i -consistency algorithm described here applies a brute force method to trim all sets S of variables of size $(i - 1)$ that cannot be extended with values from *all* variables not in S . Note that the relation R_S may be a universal relation to start with when it is not mentioned explicitly.

It is clear that we have moved beyond binary relations and have started inspecting larger compound labels for consistency. In fact even when we are looking at node consistency and arc consistency, we may

have to look at relations of larger arity if we are to implement 1-consistency and 2-consistency respectively. Let us look at an example to see why this is so. Consider the following CSP problem.

Problem CSP5

$$(\{w, x, y, z\}, D_w = D_x = D_y = D_z = \{r, b, g\}, \{(\{x, y, z\}, R_{xyz} = \{\langle r, b, g \rangle\}), (\{w, y, z\}, R_{wyz} = \{\langle b, b, g \rangle\})\})$$

Now consider the notion of 3-consistency which says that any consistent assignments to two labels can be consistently extended to three labels. Consider now an assignment $\{y = r, z = r\}$. Clearly, there is nothing inconsistent about it, since there is no explicit constraint on $S = \{y, z\}$. But it cannot be consistently extended either to w or to x . Yet our path consistency algorithm will not catch this inconsistency because it was written only for binary relations. Now, if we were to extend our path-consistency algorithm to 3-consistency then we would be forced to look at all pairs of variables and select values that can be consistently extended. This would result in the following relation pruning.

$$R_{xy} = \{\langle r, b \rangle\}, R_{xz} = \{\langle r, g \rangle\}, R_{wy} = \{\langle b, b \rangle\}, R_{wz} = \{\langle b, g \rangle\}, R_{yz} = \{\langle b, g \rangle\}$$

Now, one can see that the value $x = b$ or $x = g$ cannot be consistently extended to R_{xy} and thus must be pruned. Likewise, for the other variables, and when we apply 2-consistency we get,

$$D_w = \{b\}, D_x = \{r\}, D_y = \{b\}, D_z = \{g\}$$

We can define generalized notions of arc consistency with respect to relations of higher arity as follows (Dechter, 2003).

Given a CSP (X, D, C) with $R_S \in C$, a variable $x \in S$ is said to be arc consistent relative to R_S , only if for every value of $a \in D_x$ there exists a tuple $t \in R_S$, such that $x = a$ is in that tuple. A relation R_S is said to be arc consistent *iff* all its variables are arc consistent. A relation can be made arc consistent by reducing the domains of its variables to those values that are specified in the relation.

9.4.3 Propagation = Inferencing Relations

Given a CSP (X, D, C) , one can view the constraint propagation process as inferencing new constraints to get a new equivalent CSP (X, D, C') where $C' = C \cup \text{NewConstraints}$. Given that there are n variables in the CSP, we can have one relation of arity n , n relations of arity $(n - 1)$, nC_2 relations of arity $(n - 1)$, and so on with n relations of arity 1. One can say that the task of solving the CSP is to unearth the relation on n variables, often referred to as $R(\rho)$, the *solution relation*.

Only some of these relations may be specified in the set C of constraints, and that too only partially, containing more tuples in the relation than actually allowed. When they are not specified explicitly, one can assume them to be universal relations. One can view the process of consistency enforcement as a process of tightening the constraints of appropriate arity, including arity 1. If at any time during the propagation process, any relation becomes empty, it means that there is no solution and the CSP is an inconsistent CSP .

When a relation is implicit then constraint propagation may still reduce tuples from its domain. One has to make that relation explicit now, and hence we can view the propagation process as having inferred a new relation.

Enforcing node consistency and arc consistency can be seen as introducing new unary constraints that restrict the domains of certain variables. Traditionally, we see this as reducing the domain of a variable, say D_x , but we can also view this as having inferred a relation R_x on that variable.

Likewise, enforcing path consistency, or the generalized arc consistency (i.e. 3-consistency) can be viewed as inferring a new binary relation. Consider the following binary CSP on three variables.

Problem $CSP7$

$$(X, D, C) = (\{x_1, x_2, x_3\}, \{\{r, b\}, \{r, b\}, \{r, b\}\}, \{\{\{x_1, x_2\}, x_1 \neq x_2\}, \{\{x_2, x_3\}, x_2 \neq x_3\}\})$$

Enforcing path consistency implies creating a new relation R_{13} with scope $\{x_1, x_3\}$,

$$R_{13} = \{(r, r), (b, b)\}$$

Often, specially when enforcing higher order consistency, one may represent the complement of a relation rather than the relation because the complement is explicitly known and usually smaller. The complement of the relation is known as a *nogood*, and captures those tuples that are *not* allowed. The nogood inferred for the above problem would be,

$$R'_{13} = \{(r, b), (b, r)\}$$

Enforcing i -consistency may introduce nogoods, or equivalently relations, of arity $(i - 1)$. This means that some tuples from the relation of arity $(i - 1)$ may be removed. If that happens then the CSP may lose $(i - 1)$ -consistency, even if it was $(i - 1)$ -consistent earlier. The example in Figure 9.21 shows a case when enforcing path consistency on a three variable binary, CSP disrupts arc consistency that was there earlier. Remember that algorithm PC-2 removes only relations or edges, and the corresponding nodes still remain in the domains.

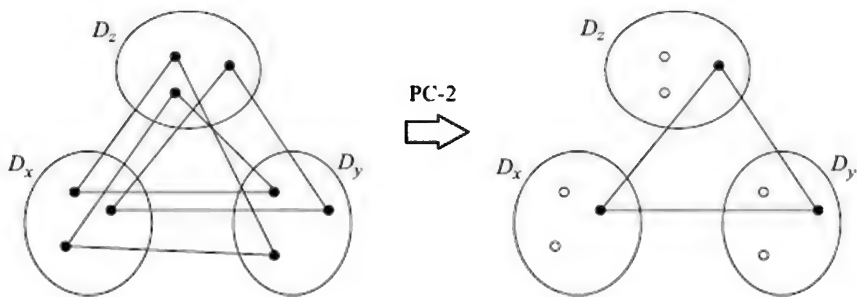


FIGURE 9.21 Enforcing path consistency on an arc consistent network may disrupt arc consistency. The matching diagram on the left is arc consistent but not path consistent. Applying the algorithm PC-1 or PC-2 gives us the diagram on the right that is path consistent but not arc consistent.

We leave the reader with the following question. If one wants to achieve strong k -consistency then will the enforcement of decreasing orders of consistency do the job?

Achieving more and more consistency reduces the amount of effort a search algorithm will have to put in. If the constraint network has n variables, achieving strong n -consistency will obviate the need for search altogether. This is because at every stage after having chosen a set of $(i - 1)$ consistent values for the first $(i - 1)$ variables, the procedure will not reach a dead end because i -consistency ensures that a consistent value for the i^{th} variable exists in the domain. But as i increases, achieving i -consistency becomes more and more expensive. In some situations, it may be profitable to work with a limited amount of consistency enforcement, and spend some time doing search.

One such situation is when the order of selecting variables for an assignment is pre-decided. Given that the search algorithm is going to inspect variables in a fixed order one need not enforce consistency between variables in both directions. In the next section we explore how much consistency needs to be enforced, based on the topology of the constraint graph.

9.5 Directional Consistency

A constraint network is said to be 2-consistent, if for every assignment of a variable there exists an assignment for *every* other variable such that the new variable is consistent with the known constraints. This ensures that *any* variable that we pick for assignment can be followed up picking any other variable. Higher order consistencies extend this notion to sets of variables. A network is said to be i -consistent, if *any* set of $i - 1$ variables is assigned a compound label then any other variable can be assigned a consistent value. However, if the order of assigning variables is known in advance, a weaker notion of consistency suffices. This weaker notion of consistency is *directional consistency*, which requires only variables chosen earlier in the assignment order to be consistent

with later ones. In the discussion that follows, we assume that an ordering of variables (x_1, x_2, \dots, x_n) has been specified, and x_1 is the first variable to be assigned a value from its domain.

Consider the map colouring CSP shown in Figure 9.22 on the left, along with two orderings ($ABCDE$) and ($EDCBA$) on the right. The domain of all the variables is $\{r, b, g\}$. Let us assume that algorithm for solving the CSP assigns values to the variables in the two given orders, and at each stage chooses a value consistent with the value assigned earlier to every related variable, called its *parent*, if any. For the order ($ABCDE$), the parent of B is A , parents of E are B, C and D . Observe that for the second ordering ($EDCBA$), every node has only one parent. The reader is encouraged to ponder over the orderings and ask which is better.

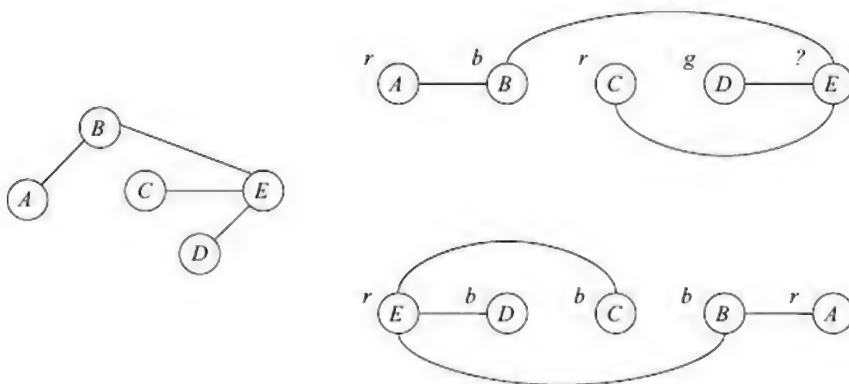


FIGURE 9.22 A map colouring problem with the domain of all variables being $\{r, b, g\}$. The figures on the right show two different orderings and a possible colour assignment proceeding from left to right. As one can see, the lower ordering colours the graph automatically, but the upper one can reach a dead end.

One possible assignment sequence for the order $ABCDE$, as shown in the figure, is,

$$A = r, B = b, C = r, D = g$$

However at this point, it is not possible to assign a value to E because the three parents of E have used up the three colours. The algorithm will need to backtrack and try another value for D . The reader should verify that for the order $EDCBA$, the assignment process goes smoothly from start to finish, and there is no need to backtrack. In fact, the solution displayed in the figure uses only two colours.

The given CSP is arc consistent, and also path consistent. What is different in the two examples is the ordering of the variables. The orderings can be characterized by a property called *width* defined as the largest number of parents any node has in that ordering. The ordering $ABCDE$ has width 3, while the ordering $EDCBA$ has width 1. The width of a given graph is the minimum of the widths of all its orderings.

A related property is called *induced width*. For a given graph, the

induced width is the width of the *induced graph*. The induced graph, for a given ordering, is the graph produced by connecting all the parents of every node to each other.

It turns out that the induced width is an indicator of how much consistency enforcement is needed to have backtrack free search. Specifically, if the induced width of a graph is $(i - 1)$ for an ordering, then i -consistency is needed to make search on that ordering backtrack free. What is more, given that we have a fixed order of selecting variables, we only need to ensure that any value we pick is consistent only with later variables. The later variables do not have a consistency requirement with respect to the earlier variables, which have already been assigned values. Thus consistency needs to be enforced only in one direction.

Clearly, it is profitable to find an ordering that has the minimum induced width. We look at some algorithms for arriving at this ordering a little later in this chapter. First, we look at the algorithms for directional consistency enforcement for a given ordering. We begin by looking at the algorithm *Directional Arc Consistency (DAC)*.

9.5.1 Directional Arc Consistency

Given an order (x_1, x_2, \dots, x_n) in which the variables of a CSP will be picked for assignment, directional arc consistency specifies that for every value a that one picks for a variable x_j , for every variable x_k such that $j < k$ that is related to x_j , there exists a value b in its domain such that $\langle a, b \rangle \in R_{jk}$. If this is not the case, the algorithm for enforcement removes the value " a " from the domain D_j of x_j .

If a value is removed from the domain of a given variable x_j then it can only affect arc consistency for a variable x_i if $i < j$. Therefore, if we enforce arc consistency from the last variable to the first, only one call to *Revise* needs to be made for every pair of related variables. The algorithm DAC is given in Figure 9.23 below.

```

DAC ( $X = (x_1, x_2, \dots, x_n), D, C$ )
1  for  $j \leftarrow n$  downto 2
2      do for each  $i < j$  s.t.  $(\{i, j\}, R_{ij}) \in C$ 
3           $D_i \leftarrow \text{Revise}(D_i, D_j, R_{ij})$ 
4  return  $(X, D, C)$ 

```

FIGURE 9.23 The algorithm DAC takes as input a CSP with an ordering on the variables and returns an equivalent CSP that is arc consistent with respect to that ordering.

The algorithm inspects $O(n^2)$ pairs of variables, but makes exactly e calls to *Revise*, where e is the number of constraints. In each call, it does $O(k^2)$ amount of work where k is the size of each domain. The complexity of DAC is therefore $O(ek^2)$.

Consider the following CSP with the same constraint graph as in Figure 9.22, but with different domains and constraints.

$$D_A = \{r, b, v\}, D_B = \{g, b, v\}, D_C = \{r, b, m\}, D_D = \{m, v, b\}, D_E = \{r, b, m\}$$

and the new relations as,

$$A = B, B = E, C = E \text{ and } D = E$$

The reader should verify that after full arc consistency is enforced, the new domains are,

$$D_A = D_B = D_C = D_D = D_E = \{b\}$$

However, when we do directional arc consistency with respect to the order *EDCBA*, we get the domains as,

$$D_E = \{b\}, D_D = \{m, v, b\}, D_C = \{r, b, m\}, D_B = \{b, v\}, D_A = \{r, b, v\}$$

Observe that the only value in D_E is b . The equality relation allows only the assignment of value b to the rest of the variables, which is indeed present in all domains. Thus, we have backtrack free search. If we had chosen the other ordering *ABCDE*, we would have got the domains as follows,

$$D_A = \{b\}, D_B = \{b\}, D_C = \{r, b, m\}, D_D = \{m, b\}, D_E = \{r, m, b\}$$

Now if we choose the assignments $A = b, B = b$ and $C = r$, which is allowed because C is not related to either A or B , we are at a dead end, and will have to backtrack. This is consistent with what happened in the map colouring example. The ordering *EDCBA* makes the graph of width 1, and induced width⁴¹, and it suffices to achieve directional arc consistency or 2-consistency.

9.5.2 Directional Path Consistency

We can extend the idea of *DAC* to Directional Path Consistency (*DPC*). Here too, we begin with the last variable and call *Revise-3* with every pair of variables, *both* of which are connected to it. *DPC* may prune the relation, whether implicit or explicit, between the parent nodes. But for achieving backtrack-free status (under certain conditions), one must be careful to enforce *DAC* after *DPC* to remove unsupported values from those domains as well. Consider a three variable map colouring problem,

$$D_A = \{r, b, g\}, D_B = \{r, b, g\}, D_C = \{r\}$$

and the relations as,

$$A \neq B : R_{AB} = \{(r, b), (r, g), (b, r), (b, g), (g, r), (g, b)\}$$

$$B \neq C : R_{BC} = \{(b, r), (g, r)\}$$

and

$$A \neq C : R_{AC} = \{(b, r), (g, r)\}$$

Assuming the order to be *ABC*, achieving *DPC* would remove all tuples containing *r* from R_{AB} . But the value *r* is still in the domains D_A and D_B , and can potentially be selected leading to a dead end. These can be removed by *DAC*. In this example, the desired effect can be achieved by applying *DAC* before *DPC* too, but the reader is encouraged to construct an example in which it is *necessary* to do so after *DPC*.

Every time a call is made to *Revise-3*, it may prune the relation between the calling pair of variables. If the relation is implicit then it infers a new explicit relation. This new relation must be added to the constraint graph before proceeding further with *DPC*. Figure 9.24 shows a map colouring CSP with a chosen ordering⁵ of variables. The graph and three implicit relations amongst parents of variable *E* are shown on the top. Observe that after a call to *Revise-3*($D_E, R_{CD}, R_{CE}, R_{ED}$), the relation R_{CD} is introduced into the constraint graph, and after a call to *Revise-3*($D_E, R_{BD}, R_{BE}, R_{ED}$), the relation R_{BD} is introduced. Next, when algorithm *DPC* moves to process node *D*, it must be aware of these two new edges to be able to prune the relation R_{BC} if needed.

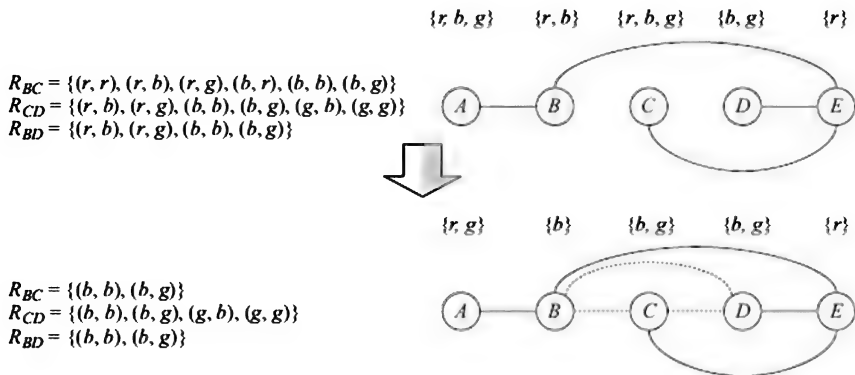


FIGURE 9.24 Another map colouring problem with a chosen ordering and the domains shown on the top. The revised domains and relations after calls to *DPC* and *DAC* at node *E* are shown below. Note that while the original relations depicted by solid edges are the not-equal relation, the new one's shown with dashed edges allow certain combinations of colour including the same ones.

The revised domains and relations after *DPC* and *DAC* are enforced and are shown below the arrow in the figure.

The algorithm *DPC* described below includes the *DAC* step.

```

DPC ( $X = (x_1, x_2, \dots, x_n)$ ,  $D$ ,  $C$ )
1 initialize  $E = \{(i, j) \mid \{x_i, x_j\} \text{ is the scope of some relation}\}$ 
2 for  $k \leftarrow n$  downto 2
3   do for each  $i < j < k$  such that  $(i, j), (j, k) \in E$ 
4      $R_{ij} \leftarrow \text{Revise-3}(D_k, R_{ij}, R_{ik}, R_{kj})$ 
5      $E \leftarrow E \cup \{(i, j)\}$ 
6      $C \leftarrow C \cup \{(\{i, j\}, R_{ij})\}$ 
7   do for each  $i < k$  s.t.  $(\{i, k\}, R_{ik}) \in C$ 
8      $D_i \leftarrow \text{Revise}(D_i, D_k, R_{ik})$ 
9 return  $(X, D, C)$ 

```

FIGURE 9.25 The algorithm *DPC* takes as input a CSP with an ordering on its variables and returns an equivalent CSP after enforcing directional path and arc consistency. It may add new edges to the constraint graph, and therefore new explicit constraints.

The complexity of *DPC* is $O(n^3k^3)$, where n is the number of variables and k the size of each domain. For the last variable in the worst case, it has to look at $(n-1) * (n-2)$ or $O(n^2)$ pairs of variables. Each pair of variables may have at most k^2 edges, for each of which k values in the domain of the last variable may have to be examined. Thus, for processing the last variable, the complexity is $O(n^2k^3)$, and processing all the n variable will give us a worst-case performance of $O(n^3k^3)$. Observe that we consider all pairs of variables instead of all edges because the procedure can add more edges during processing.

9.5.3 Adaptive Consistency

We can extend the notion of directional consistency for higher orders. If we want to enforce i -consistency, we need to look at combinations of $(i-1)$ parents of a node. For strong i -consistency, we would need to establish directional j -consistency for all $j < i$ as well.

Given the fact that directional consistency is enforced from the last to the first variable, when the time comes to process variable x_i we already know how many parents it has, and therefore we know what degree of consistency is needed for that node. This leads us to the idea of *adaptive consistency*, in which a call to an appropriate degree of i -consistency is made for each node. The following algorithm illustrates this idea of adaptive consistency. Observe that the algorithm in Figure 9.26 does not necessarily implement strong consistency.

```

AdaptiveConsistency ( $X = (x_1, x_2, \dots, x_n)$ ,  $D$ ,  $C$ )
1 initialize  $E = \{(i, j) \mid \{x_i, x_j\} \text{ is the scope of some relation}\}$ 
2 initialize  $C' \leftarrow C$ 
3 for  $k \leftarrow n$  downto 2
4    $S \leftarrow \text{Parents}(x_k)$ 
5    $R_s \leftarrow \text{Revise-I}(S, x_k)$ 
6    $E \leftarrow E \cup \{(i, j) \mid i, j \in S\}$ 
7    $C' \leftarrow C' \cup \{(S, R_{ij})\}$ 
8 return  $(X, D, C')$ 

```

FIGURE 9.26 The algorithm *AdaptiveConsistency* takes as input a CSP with an ordering on its

variables and returns an equivalent CSP after enforcing directional of appropriate degree. It may add new edges to the constraint graph, and therefore new explicit constraints. The function $Parents(x_k)$ returns the set of parents of a node and the procedure $Revise-I(S, x_k)$ is defined in Fig. 9.20.

The induced width of the ordered constraint graph will dictate the amount of work done by adaptive consistency.

In general, if the induced width is w then enforcing strong w -consistency may require one to inspect subsets of smaller size as well. In the end, while selecting the value for a variable x_i , the reduced domain of x_i should only contain values that participate in all constraints for all future variables. We argue that this happens automatically if we use the Adaptive Consistency algorithm. If a given node x_k has P parents then the algorithm ends up adding edges between all the parents, and establishing a constraint of arity $(P - 1)$. Let x_P be the first such parent the algorithm encounters when processing in the last to first order. Then x_P would have all the other parents of x_k as parents, apart from any parents of its own that it might have had. Directional consistency will make sure that only those compound labels for the parents of x_P are allowed that can be extended with a value for x_P . As we progress (backwards), the maximum possible size of the sets of parents will decrease, even though more edges are added on the way, eventually pruning the domains of independent variables. Observe that if the graph is connected then x_1 will have an edge to x_2 by the time x_2 is processed⁶. When the second variable x_2 is processed, it will prune the domain D_1 of x_1 so that only values consistent with x_2 are retained. If x_1 is also connected to some x_i then the constraint R_{12} would have been introduced (or pruned) to allow only pairs consistent with the later variable. This will constrain the choice of a value for x_2 . Likewise, a constraint R_{123} will constrain the choice a value for x_3 , and so on. We illustrate this with a couple of examples.

Consider applying adaptive consistency to the problem depicted in Figure 9.24, shown below in Figure 9.27. First, 4-consistency is applied to the variable E since it has three parents. This infers the relation R_{BCD} and adds the edges (B, C) , (C, D) and (B, D) to the constraint graph. Now the variable D has two parents and so 3-consistency is applied to it. This prunes the relation R_{BC} as shown. The edge (B, C) already exists. Then variables C and B are processed and 2-consistency is applied, pruning the domains of B and A respectively, as shown.

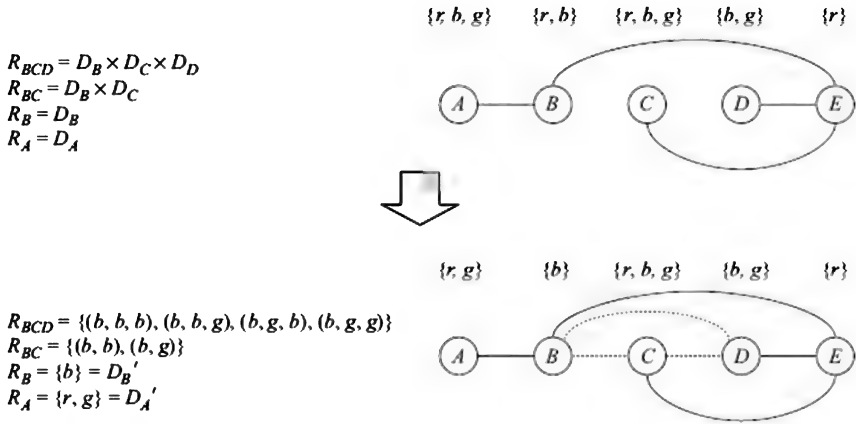


FIGURE 9.27 Adaptive consistency on the problem in Fig. 9.24 infers a different set of relations as shown. It prunes the domains of only variables B and A , when establishing consistency for C and B respectively. The resulting graph is backtrack free, when values are chosen respecting the inferred constraints.

The reader should verify that the resulting graph yields backtrack free search. There is a spurious value “ r ” in the domain D_C but it is never selected because the inferred relation R_{BC} does not allow it. Consider another problem of the five variables as follows,

Problem CSP8

- $X = \{x_2, x_2, x_3, x_4, x_5\}$
- $D = \{D_1 = D_2 = D_4 = \{r, b, g\}, D_3 = \{r, b\}, D_5 = \{b, g\}\}$
- $C = \{C_{ij} \mid x_i, x_j \in X\}$ where $S_{ij} = \{x_i, x_j\}$ and $R_{ij} = (x_i = x_j)$.

The graph is fully connected and the relation is that any two variables are assigned the same colour. Let us assume the ordering x_2, x_2, x_3, x_4, x_5 . Adaptive consistency proceeds as follows.

1. 5-consistency is applied to the variable x_5 and the relation R_{1234} is inferred as

$$R_{1234} = \{(b, b, b, b)\}$$

2. 4-consistency is applied to the variable x_4 and the relation R_{123} is inferred as

$$R_{123} = \{(b, b, b)\}$$

3. 3-consistency is applied to the variable x_3 and the relation R_{12} is inferred as

$$R_{12} = \{(b, b)\}$$

4. 2-consistency is applied to the variable x_2 and the relation R_1 is inferred as

$$R_1 = \{b\}$$

Observe that no domain is pruned explicitly, except that the value of

variable x_1 is restricted to “b” because of R_1 . Subsequently, because of the inferred domains, only the value “b” is selected for all other variables, in a backtrack free manner. See also Problems 14 and 15 for more examples of adaptive consistency.

9.5.4 Minimum Width Orderings

It is clear that finding an ordering with the minimum induced width is beneficial from the complexity perspective. It turns out that while finding the minimum width can be done easily, finding a minimum induced width ordering is an NP-complete problem.

The following greedy algorithm yields a minimum width ordering of the nodes of a graph. The algorithm constructs the ordering from the last to first, at each stage choosing a variable with the smallest number of neighbours in the constraint graph, and removing it along with its edges from the graph. The algorithm is depicted in Figure 9.28. It returns a list which captures ordering of the nodes. Observe that the index k in the loop is not used. The *for* loop is only suggestive of the fact that the order is constructed last to first. One could have used a *while* or repeat-until loop as well.

```

MinWidth (G = (V = {v1, ..., vN}, E))
1  O ← ( )
2  for k ← n downto 1
3    lowestDegree ← RemoveLowestDegreeNode(G)
4    E ← E \ {(x, y) | x = lowestDegree or y = lowestDegree}
5    O ← cons(lowestDegree, O)
6  return O

```

FIGURE 9.28 The algorithm *MinWidth* takes a graph with N vertices and returns a minimum width ordering of the vertices. We assume a function *RemoveLowestDegreeNode*(G) finds the vertex with the lowest number of edges emanating from it, and removes it from the set V . The function *cons* is a Lisp like function that adds an element to the head of a list.

It was observed by Freuder (1982) that if the constraint graph is a tree then one can always come up with an ordering of width one. The reader should verify that the above algorithm produces such an ordering for the problem in Figure 9.22. If one enforces directional arc consistency on such an ordering, the resulting graph will still have width one. This is due to the fact that each node has exactly one parent in this ordering, and, therefore, the induced graph is the same as the given graph. Hence, constraint networks that are trees can be made backtrack free by choosing a minimum width ordering and enforcing DAC on the resulting graph (Freuder, 1982).

In general though, enforcing an appropriate amount of directional consistency on a given ordering may increase the width of the graph. Figure 9.29 shows a graph with a minimum width ordering *ABCDEFGF* with width = 2. Enforcing directional path consistency makes the induced width of the graph 4.

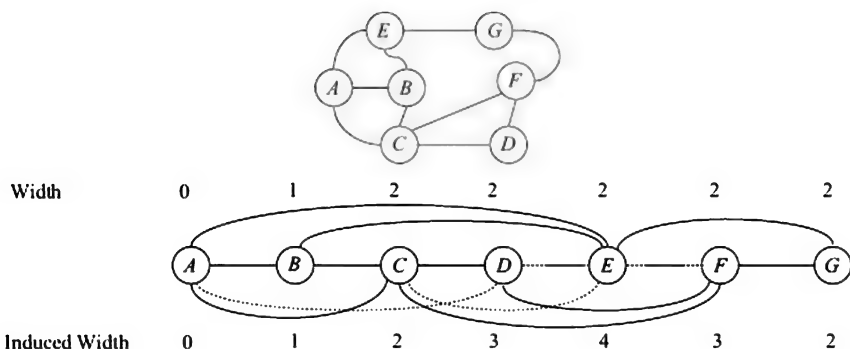


FIGURE 9.29 The ordering here is a minimum width ordering. Enforcing higher order consistency on a minimum width graph is likely to increase the width. The width of the example graph here increases from 2 to 4 as shown.

While it is hard to find the minimum induced width orderings, one can adapt the *MinWidth* algorithm to account for connectivity of parents of a node. The two algorithms described below produce good orderings. The algorithm *MinInducedWidth* described in Figure 9.30 chooses a node with the smallest number of neighbours in the constraint graph, and removes it and the edges emanating from it. Before proceeding further, however, it connects the parents of the node just selected.

```

MinInducedWidth ( $G = (V = \{v_1, \dots, v_N\}, E)$ )
1  $O \leftarrow ()$ 
2 for  $k \leftarrow n$  downto 1
3    $lowestDegree \leftarrow RemoveLowestDegreeNode(G)$ 
4    $E \leftarrow E \cup \{(p, q) \mid (p, lowestDegree) \in E \text{ and } (lowestDegree, q) \in E\}$ 
5    $E \leftarrow E \cup \{(q, p) \mid (q, lowestDegree) \in E \text{ and } (lowestDegree, p) \in E\}$ 
6    $E \leftarrow E \setminus \{(x, y) \mid x = lowestDegree \text{ or } y = lowestDegree\}$ 
7    $O \leftarrow cons(lowestDegree, O)$ 
8 return  $O$ 

```

FIGURE 9.30 The algorithm *MinInducedWidth* is like the algorithm *MinWidth* in Fig. 9.28, except that while removing the selected node from the graph it connects its parents with edges. Observe that we have added two pairs (p, q) and (q, p) for every new edge for simplicity.

Figure 9.31 shows the ordering produced by the algorithm *MinInducedWidth* on the constraint graph depicted in Figure 9.29. Observe that the induced width on this ordering *ABCEFDG* is three, which is lower than taking a minimum width ordering and then connecting the parents.

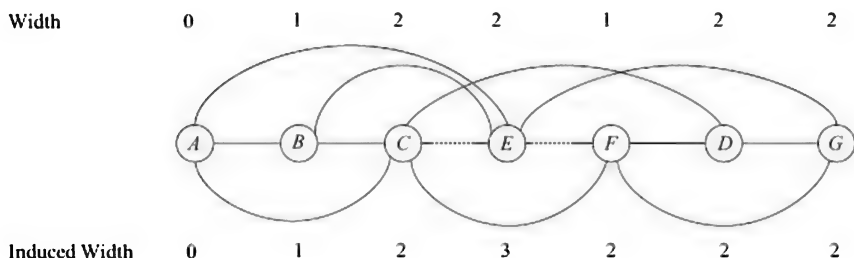


FIGURE 9.31 The induced width of the ordering produced by the algorithm *MinInducedWidth* on the graph of Fig. 9.29 is three.

Another algorithm that has been experimentally shown to be better than the *MinInducedWidth* algorithm is the *MinFill* algorithm. The *MinFill* algorithm selects that node to be placed in the reverse order for which the number of edges needed to connect all its parents is the minimum.

9.6 Algorithm Backtracking

The constraint propagation methods described above are directed towards shrinking the space in which one searches for a solution. It is possible to enforce consistency to the extent that finding a solution can be done in one pass over the variables in a backtrack free manner, selecting an allowed value for each variable. But such reasoning has a computational cost that may become prohibitive. At the other end is the conceptually simple search algorithm that tries all combinations of values. This too may be computationally too expensive. Somewhere in between lie hybrid approaches that combine search and reasoning, and one has to often find a trade off between the amount of search and propagation an algorithm does.

The simplest search algorithm is one that tries all combinations for all values of all variables looking for a solution. Such an algorithm could be implemented as *Depth First Search* (see Chapter 2). If we implement the algorithm in Figure 2.16 directly, it will test for consistency (the *goalTest* procedure) only when it has selected values for *all* variables. This is a waste of effort since we have the constraints between subsets of variables explicitly available. The algorithm *Backtracking* described below tests for applicable constraints while selecting values for each variable. If it cannot find a consistent value for a variable then it backtracks at that point itself. In the discussion below, we assume that the variables are named x_1, x_2, \dots, x_n , and their domains as D_1, D_2, \dots, D_n . We assume a procedure *SelectValue* that selects and returns a value from the domain of the j^{th} variable that is consistent with the values for the earlier variables chosen. The solution is assembled in the reverse order in an assignment list A . The list A is initialized to a list containing an empty list to prevent a call to the function *Tail* with an empty list.

The algorithm *Backtracking* makes copies of the domains of all variables and the algorithm *SelectValue* extracts values out from the copied domain. Note that the copied domain is shrunk inside the *SelectValue* function, implying that the domain is treated as a global variable, or the call is call-by-name. This is necessary if the algorithm has to backtrack and try another value. Assuming that the domains are represented as lists, the algorithm in Figure 9.32 simply chooses the value at the head of the list. Later, we will look at variations that apply some reasoning to filter values not likely to succeed.

```

Backtracking ( $X, D, C$ )
1  $A \leftarrow (())$ 
2  $i \leftarrow 1$ 
3  $D_i' \leftarrow D_i$ 
4 while  $1 \leq i \leq n$ 
5   do  $a_i \leftarrow \text{SelectValue}(D_i', A, C)$ 
6     if  $a_i = \text{null}$ 
7       then  $i \leftarrow i - 1$ 
8          $A \leftarrow \text{Tail}(A)$ 
9     else
10       $A \leftarrow \text{Cons}(a_i, A)$ 
11       $i \leftarrow i + 1$ 
12      if  $i \leq n$ 
13        then  $D_i' \leftarrow D_i$ 
14 return  $\text{Tail}(\text{Reverse}(A))$ 

SelectValue ( $D_i', A, C$ )
1 while not empty( $D_i'$ )
2   do  $a_i \leftarrow \text{Head}(D_i')$ 
3      $D_i' \leftarrow \text{Tail}(D_i')$ 
4     if Consistent( $A, x_i = a_i$ )
5       then return  $a_i$ 
6 return null

```

FIGURE 9.32 The algorithm *Backtracking* explores the domains in a depth first manner. However at each stage, it calls the procedure *Consistent*(A, a_i) to check that the value being chosen is consistent with the compound label assembled so far. We also assume the list processing functions *Head*, *Tail* and *Cons*. Initializing A with a list containing one element, in this case the empty list, is simply to prevent a call to *Tail* with an empty list. The *return* statement is modified to leave out this element. It returns an empty list if there is no solution.

If *SelectValue* cannot find a value, it returns “null” which prompts the calling procedure to backtrack to look for another value for the previous variable from the values remaining in the copy of its domain (steps 7 and 8). Observe that if an appropriate amount of (directional) consistency has been enforced before calling algorithm *Backtracking* there would be no backtracking. *Backtracking* to the previous choice point (variable) is termed *chronological* backtracking. A little later we look at approaches to jumping back to variables that may have been the cause of inconsistency that resulted in the backtracking. Observe that every time it moves *forward* to the next variable (steps 11 and 12), it makes a fresh copy of the domain.

The algorithm *Backtracking* can abstractly be described as follows.

- Loop: 1. Select a variable.
2. Select a value for the variable.
- 2a. If none then undo some choices and reselect a

previous variable.
 2b. Else go to Loop.

Steps 1, 2 and 2a contain the three choices the algorithm has to make. Each of them offers the possibility of making an informed choice. The choice at step 1 can be made such that some critical variables are assigned values early in search. We have already seen that the motivation for choosing a minimum induced width ordering is to arrange the variables such that future variables are constrained by as few parents as possible. In the next section, we will also see how the choice of variables is influenced dynamically by the current domain sizes of future variables.

Having chosen a variable, the choice of a value for that variable at step 2 determines how future variable domains are affected. Some choices may leave more options for future variables. In the next section we look at varying degrees of lookahead that an algorithm can do to improve the choice at step 2.

If the algorithm has reached a dead end, then in step 2a it needs to go back and try another value for an earlier variable. But the question is, which earlier variable? The inconsistency that caused the dead end could have been due to the choice of any variable assigned values in the past. Look-back approaches are designed to make this choice in an informed manner. We will look at them in Section 9.8.

We will illustrate the algorithms with the CSP in Figure 9.33, which is a map colouring CSP on the constraint graph of Figure 9.29 shown here with the domains. The ordering chosen is *GDBFEAC*, which the reader would have noticed is not the best one, though it is useful for illustrating the algorithms.

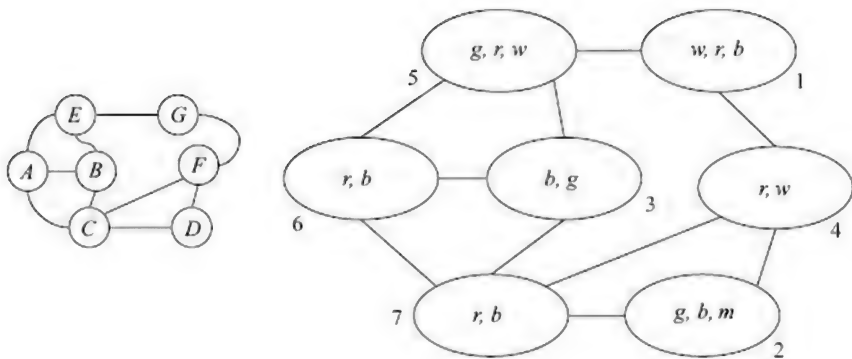


FIGURE 9.33 A CSP associated with the constraint graph of Figure 9.29. We assume the problem is a map-colouring problem with the domains as shown. The fixed ordering chosen is *GDBFEAC*, depicted by numbers in the graph on the right.

The search tree generated by algorithm *Backtracking* is shown in Figure 9.34. Each node, except the root, represents a choice of value for a variable. The value has to be consistent with the values of its ancestor

procedures we discuss, prune the domains of *future* variables during the process of selecting a value for the variable. They reject the value being considered for the variable if it can be determined that some future domain has become empty.

The domains have to be carefully maintained. The procedure for selecting values should undo any changes to future domains it has made before finding the current value unfit. Likewise, the main procedure must also undo such pruning if it has to backtrack from a given variable. In the programs below, we assume that the domains and any copies that we make are *global* and *visible* to all procedures. In practice, one could package the needed domains into a structure and pass them as a parameter.

We use a generalized lookahead algorithm that will call different versions of *SelectValue-X* for varying degrees of propagation (in the style of (Dechter, 2003)). The algorithm is described in Figure 9.34. Since the *SelectValue-X* procedure will prune future domains, removing values variously inconsistent with the value being considered, the main algorithm needs to be able to restore the values if it backtracks. In the algorithm below, this is done by maintaining copies of *all* domains at *each* level in the search tree. Steps 11–13 in the algorithm restore domains on backtracking to as they were, before this variable was processed. Steps 17–19 make copies when advancing to the next variable.

The simplest consistency check is to make sure that no domain of a future variable is made empty. This was illustrated earlier with the 6-queen problem in Figure 9.3. The procedure *SelectValue-FC* is described below.

```

Backtracking-with-LookAhead (X, D, C)
1  A ← (( ))
2  for i ← 1 to n
3      do  $D_{\neg i} \leftarrow D_i$ 
4           $D_{\neg i} \leftarrow D_{\neg i}$ 
5  i ← 1
6  while  $1 \leq i \leq n$ 
7      do  $a_i \leftarrow \text{SelectValue-X}(i, A, C)$ 
8          if  $a_i = \text{null}$ 
9              then i ← i - 1
10                 A ← Tail(A)
11                 if  $1 \leq i$ 
12                     then for k ← i+1 to n
13                         do  $D_{ik} \leftarrow D_{(i-1)k}$ 
14                 else
15                     A ← Cons( $a_i$ , A)
16                     i ← i + 1
17                     if  $i \leq n$ 
18                         then for k ← i to n
19                             do  $D_{ik} \leftarrow D_{(i-1)k}$ 
20  return Tail(Reverse(A))

```

FIGURE 9.35 Backtracking with *Lookahead* prunes domains of forward variables. When it backtracks, it needs to undo the pruning done at current level. For this, it keeps main domains for each variable at each level, and restores pruned domains for future variables when it backtracks. The call to *SelectValue* does not have domains as parameters since we have assumed them to be globally visible.

9.7.1 Forward Checking

In the previous example in Figure 9.34, the variable C has two values in the domain. Of these, $C = b$ gets precluded when $B = b$ is assigned, and $C = r$ gets precluded when $A = r$ is assigned. The algorithm *Backtracking* would see this, only when the turn of variable C comes. The forward checking procedure would not have chosen the second assignment $B = b$ because it removes values from future domains that are directly conflicting with the value being considered and rejects the value if any future domain becomes empty due to that. The algorithm *SelectValue-FC* in Figure 9.36 starts off by backing up copies of domains D_{ik} , $i < k \leq n$ (steps 1–2). The domain D_{ij} contains values for x_i that have not been pruned by earlier variables. The algorithm picks one value and prunes inconsistent values from future domains (steps 6–16). At any point, if any domain becomes empty the algorithm aborts and resets domains it has pruned (steps 12–15), and tries another value (steps 3–6). If after pruning all domains they still are non-empty, the algorithm returns the selected

value for x_i . Observe that the domains D_{ik} , $i < k \leq n$ that are globally visible may have been pruned in the process, and will be copied when the algorithm *Backtracking-with-LookAhead* will move to the next variable (steps 12–13 of Figure 9.35).

```

SelectValue-FC (i, A, C)
1  for k ← i+1 to n
2    do  $C_{ik} \leftarrow D_{ik}$ 
3  while not empty( $D_{ii}$ )
4    do  $a_i \leftarrow \text{Head}(D_{ii})$ 
5        $D_{ii} \leftarrow \text{Tail}(D_{ii})$ 
6    notEmptyDomain ← true
7    k ← i + 1
8    while notEmptyDomain and k ≤ n
9      do for all b ∈  $D_{ik}$ 
10         if not Consistent(A,  $x_i = a_i$ ,  $x_k = b$ )
11            then remove b from  $D_{ik}$ 
12         if Empty( $D_{ik}$ )
13            then notEmptyDomain ← false
14            for j ← i+1 to k
15               do  $D_{ij} \leftarrow C_{ij}$ 
16         else k ← k+1
17  if notEmptyDomain
18     then return  $a_i$ 
19  return null

```

FIGURE 9.36 The algorithm *Backtracking* explores the domains in a depth first manner. However at each stage, it calls the procedure *Consistent*(A, a_i) to check that the value being chosen is consistent with the compound label assembled so far. We also assume the list processing functions *Head*, *Tail* and *Cons*.

We look at the algorithm working on the problem in Figure 9.33. Figure 9.37 below depicts the matching diagram for the CSP.

Processing the nodes in the given order *GDBFEAC*, algorithm *Backtracking-with-LookAhead* begins by calling *SelectValue-FC* with variable *G*. *SelectValue-FC* assigns *w* to *G*, and prunes the value *w* from the domains of *E* and *F*, the variables that *G* participates in constraints with. Next, the variable *D* is taken up, and the value *g* selected for it. This has no effect on other variables. After that value *b* is selected for *B*, and is therefore removed from the connected domains *A* and *C*. The matching diagram at this stage is shown in Figure 9.38.

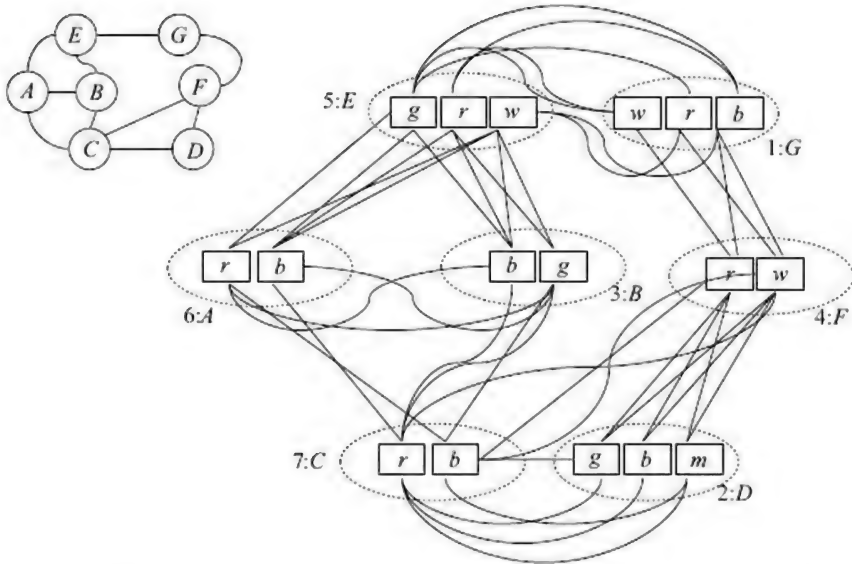


FIGURE 9.37 The matching diagram associated with the CSP of Fig. 9.33. The chosen ordering and the name of each node is shown next to each node. The algorithm begins by picking value w for variable G .

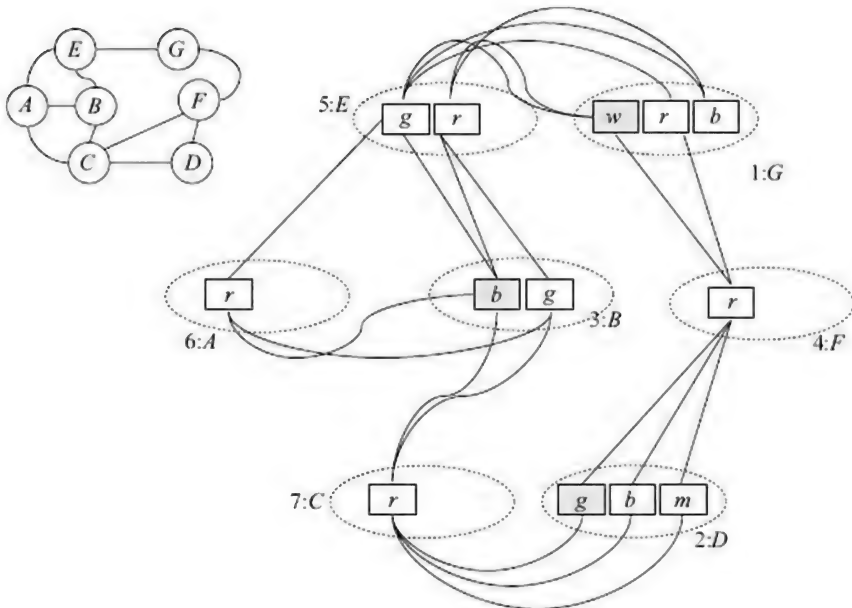


FIGURE 9.38 Forward Checking. After $G = w$, $D = g$, $B = b$, *Forward Checking* does not notice that AC and CF have become arc inconsistent. It carries on to F . It will try to pick value $F = r$ and remove r from the connected domains G and C . It will now see that the domain of C has become empty. Since there is no other value left in the domain of F , it will backtrack to B and try a new value g .

At this point, the algorithm attempts to pick a value for the variable F . There is only one value, r , in its domain at this stage. The forward-checking algorithm will remove values not consistent with ($G = w$, $D = g$, $B = b$, $F = r$) from the domains of the future variables E , A , and C . This results in the value r being removed from the domain of C . The domain of C becomes empty and the test in line 12 of the algorithm (Figure 9.36) succeeds and the value $F = r$ is abandoned. The domain of F , D_{44} , has become empty by now and the algorithm *SelectValue-FC* returns *null*, prompting *Backtracking-with-LookAhead* to backtrack to the previous variable. The reader is encouraged to continue tracing the progress of *Forward Checking* and verify that the search tree generated is smaller than the one generated by *Backtracking* (Figure 9.34).

Forward Checking looks ahead to check that there exists a value in each future domain that is consistent *with the value being considered*. It does not check whether future domains are consistent (in our case, arc consistent) with *each other*. For example, when the value $B = b$ has been chosen, the variable C is no longer arc-consistent with both A and F . An algorithm that explores the relation amongst future variable might be able to see that and, as a result, not choose the assignment $B = b$ in the first place. Various degrees of lookahead have been investigated and some of the well known variations are described below.

9.7.2 Arc Consistency Lookahead

The AC lookahead or *Full Lookahead* implements full arc consistency between the remaining variables. It considers all pairs of future variables and removes values from one domain for which a consistent value in the other domain does not exist. The *Directional Arc Consistency Lookahead*, or *Partial Lookahead*, does only directional arc consistency for the future variables. In both cases, the values assigned to existing variables are used for checking consistency. We leave the algorithms for full and partial lookahead as an exercise for the reader (see also (Haralick and Elliott, 1980)). Here we trace the progress of the *Full Lookahead* algorithm on the problem in Figure 9.33. Like *Forward Checking*, the algorithm *Full Lookahead* too begins by trying $G = w$, and removing w from its connected variables E and F . But removal of these two values triggers a chain of value removals (in the manner of algorithm AC-3), resulting in the matching diagram shown in Figure 9.39. Observe that the value r has not been pruned from the domain of G because G is not a future variable and does not participate in the propagation. Full lookahead can see that choosing $G = w$ will result in a dead end in which the fifth variable E will have no consistent assignment, and will abandon $G = w$ right there.

Full Lookahead avoids picking the value w for the variable G . The reader should verify that it will proceed with $G = r$, $D = g$ and try $B = b$, at which point, it will be able to foresee that one of A or C will have their domain empty. It will refuse to assign $B = b$ and try $B = g$ next. This will

lead to the solution shown in Figure 9.34, but in a backtrack free manner. The reader is encouraged to draw the matching diagram at the point when the algorithm finds the solution.

In the above example, full lookahead resulted in a backtrack free search, but in general one may have to check for higher order consistency, while doing the lookahead to have backtrack free search. In practice, the cost of doing this may be prohibitive and one may have to trade-off search effort with propagation effort appropriately for each domain.

Full lookahead may be useful when a variation of backtracking, in which the variables do not have a predetermined ordering, is used. Given a fixed ordering, one can use the less computationally expensive *Partial Lookahead* that does DAC on future variables.

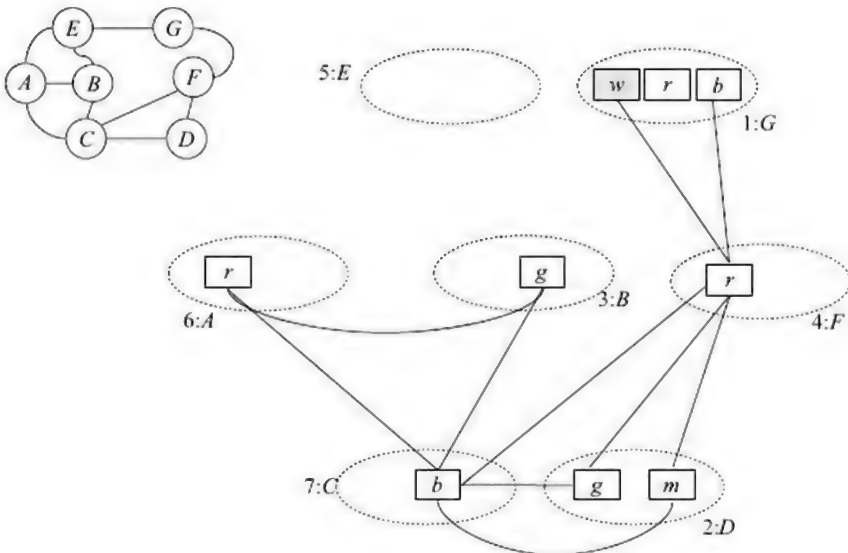


FIGURE 9.39 Full Lookahead $G = w$. Remove w from F and E . Remove r from C , because it is not arc consistent with F . Next, remove b from A , B , and D because not consistent with C . Next, remove g (not consistent with B) and r (not consistent with A) from E . At this point, domain of E has become empty and therefore $G = w$ is rejected.

9.7.3 Value and Variable Ordering

The algorithms described above use a predetermined ordering of variables, and for each variable, choose the first value that is allowed by whatever degree of lookahead being done. One can make a more informed decision for both these choices.

Instead of choosing the first past-and-future-consistent value that we find for a variable, we could investigate all values in a domain (using a limited degree lookahead) and choose the value for instantiation based on some heuristic criteria (Dechter, 2003).

One heuristic called *min-conflicts* chooses that value that does the least amount of pruning of future domains. Another heuristic, *max-domain-size*, strives to keep the future domains as large as possible. For each candidate value, it computes the size of the smallest resulting domain, and chooses that value for which the size (of the smallest future domain) is largest. A third heuristic, *estimated-solutions*, takes a more global view and looks at the sizes of all remaining domains, by taking their product. The idea is that the product represents the maximum number of possible solutions remaining after the choice of a given value for a variable.

The minimum width and the minimum induced width ordering studied earlier in this chapter were based on the topology of the underlying constraint graph. They were oblivious of the domains of the variables. The number of values that the domain of a variable has determined the topology of the search tree lying below the variable. The number of subtrees below the variable is equal to the number of values in the domain of the variable. Thus, if we choose an ordering in which the variables of smallest domains are chosen first, we will get a search tree with the smallest number of nodes, as illustrated in Figure 9.40. In the example the domains are $D_A = \{w, r, b\}$, $D_B = \{g\}$ and $D_C = \{b\}$.

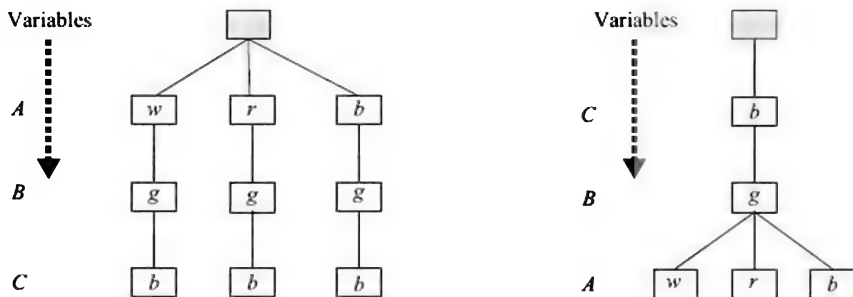


FIGURE 9.40 The number branches below a node depends upon the number of values in the domain of the variable. Choosing smaller domain variables first, results in smaller trees, as in the tree on the right.

The domain sizes of variables change dynamically when lookahead algorithms prune domains of future variables. It is possible that some variables may be pruned more than others. In such a situation, one may want to investigate the domains of the future variables before choosing the variable to instantiate next. The high level algorithm for backtracking search with dynamic variable ordering would look like the following.

- Loop:
- Choose the variable with the smallest domain.
 - Choose a value for the variable extending the solution vector A .
 - Prune the domains of all future variables, using lookahead w.r.t. A .

Refinement of the high level algorithm will fill in the method for selecting the value for the selected variable, backtracking if no value is found, and exiting with a solution vector or a failure message as the case may be. Observe that if the value for the variable is selected using *Forward Checking* or *Full Lookahead* then the domains of all future variables would have been pruned during the selection process itself.

9.8 Strategic Retreat

In the algorithms that we have discussed above, and the search methods in the earlier chapters, when a partial solution cannot be extended, the algorithm backtracks to the *latest* choice point and tries another choice. This mode of backtracking is called *chronological* backtracking. In a way, this is a blind (and conservative) strategy that is devised to explore the complete space in a simple and systematic manner. However, as we saw in Chapter 6 (see Figure 6.1), a search algorithm will often repeatedly explore a part of the search space in which no solution lies, a behaviour known as *thrashing*. This is also evident in the space explored by the algorithm *Backtracking* described earlier in this chapter. A cursory look at Figure 9.34 will reveal that the three subtrees located below the choice of the first variable $G = w$ are identical and without a solution. In hindsight, it is clear that given $G = w$ (or Evening = Visit_Mall⁷ in Chapter 6), it is futile to explore different combinations of the *remaining* choices. The question is, can an algorithm have the foresight to skip some of this exploration destined to fail?

When the search algorithm reaches a dead end, it needs to undo some of the decisions made and try something different. In the context of CSP, let us say that the algorithm has constructed a compound label $a_k = (x_1 = v_1, x_2 = v_2, \dots, x_k = v_k)$ and is unable to find a consistent value for the $(k + 1)^{\text{st}}$ variable. We say that x_{k+1} is inconsistent with a_k , and x_{k+1} is a *dead-end* variable associated with the dead-end assignment a_k . An intelligent backtracking algorithm will try and determine the *reason* for this inconsistency and go back and undo the decision at the *culprit* variable. We often use the term *dependency-directed backtracking* for such algorithms. Let us say that the culprit variable is x_i where $1 \leq i \leq k$. We say that jumping back to try a different value for x_i is *safe*, if trying a different value for any x_j where $i < j \leq k$ cannot lead to a consistent value for x_{k+1} . This means that the compound labels a_i and a_j where $i < j \leq k$ are all inconsistent with x_{k+1} . We say that a jump to x_i is *maximal*, if jumping back to an earlier variable is not safe.

An algorithm that performs safe and maximal jumps from dead ends is the *BackJumping* algorithm.

9.8.1 Algorithm *Backjumping*

The algorithm *Backjumping* presented by John Gaschnig (1977; 1978)

keeps track of the *values* of past variables that conflict with the current variable. When a dead end occurs for the variable x_{k+1} , it identifies the culprit variable x_i as the *latest variable* whose (assigned) value is the *first to conflict with some value* of x_{k+1} . The idea is that such a variable is likely to be safe and maximal for jumping back to. We illustrate this with an example from the 6-queens problem. Let a search algorithm place five queens as shown in Figure 9.41 before it reaches a dead end. There is no value for the sixth queen. The numbers in the row 6 on the chessboard represent the earliest queen that conflicts with that square. The numbers in the row 6 on the chessboard represent the earliest queen that conflicts with that square. For example, the value 3 in the square $\langle 6, b \rangle$ says that queen 3 is the earliest one that conflicts with placing queen 6 on this square. Gaschnig's *BackJumping* algorithm employs a ratchet variable called $latest_k$ for the k^{th} variable that keeps track of the highest such values. In the figure, this value is 4 from the square $\langle 6, d \rangle$.

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>
1	♔	×	×	×	×	×
2	×	×	♔	×	×	×
3	×	×	×	×	♔	×
4	×	♔	×	×	×	×
5	×	×	×	♔	×	×
6	1	3	2	4	3	1
<i>latest₆</i>	1	3	3	4	4	4

FIGURE 9.41 Gaschnig's *BackJumping* identifies the culprit as the latest variable that has conflicted with a value being considered. In this figure, there is no value for queen 6. The numbers in row 6 identify the first queen that conflicts that value (column label). The variable $latest_6$ is a ratchet that keeps track of the highest such value.

Gaschnig's algorithm is like *Backtracking*, except that when the procedure to select a value is called for the variable x_i it either returns a value for that variable, or returns *null* along with the value for $latest_i$. In the algorithm in Figure 9.42, we assume⁸ that the variables $latest_i$ is in a global array $latest(i)$ visible to both procedures. We also assume a function $SubLabel(A, k)$ that returns a prefix of the label constituting of (chronologically) the first k values in a compound label A . When *BackJumping* calls *SelectValue-GBJ* with a variable x_i , the latter inspects values in the domain D_i one by one, till it finds a consistent value. For each value, it checks for consistency with the prefix of k values in the label, k varying from 1 to i . If for some k , it cannot find a value then it marks that k as a possible culprit. The real culprit is the highest such index found for each of the values of x_i . Note that the culprit is needed only if all values of x_i are inconsistent, and the state is a dead end.

```

Backjumping ( $X, D, C$ )
1  $A \leftarrow (( ))$ 
2  $i \leftarrow 1$ 
3  $D_i' \leftarrow D_i$ 
4  $\text{latest}(i) \leftarrow 0$ 
5 while  $1 \leq i \leq n$ 
6     do  $a_i \leftarrow \text{SelectValue-GBJ}(D_i', A, C, i)$ 
7         if  $a_i = \text{null}$ 
8             then  $i \leftarrow \text{latest}(i)$ 
9              $A \leftarrow \text{Tail}(A)$ 
10        else
11             $A \leftarrow \text{Cons}(a_i, A)$ 
12             $i \leftarrow i + 1$ 
13             $\text{latest}(i) \leftarrow 0$ 
14            if  $i \leq n$ 
15                then  $D_i' \leftarrow D_i$ 
16 return  $\text{Tail}(\text{Reverse}(A))$ 

SelectValue-GBJ ( $D_i', A, C, i$ )
1 while not  $\text{empty}(D_i')$ 
2     do  $a_i \leftarrow \text{Head}(D_i')$ 
3          $D_i' \leftarrow \text{Tail}(D_i')$ 
4          $\text{conflicting} \leftarrow \text{false}$ 
5          $k \leftarrow 1$ 
6         while  $k < i$  and not  $\text{conflicting}$ 
7             if  $k > \text{latest}(i)$  then  $\text{latest}(i) \leftarrow k$ 
8             if  $\text{Consistent}(\text{SubLabel}(A, k), x_i = a_i)$ 
9                 then  $k \leftarrow k + 1$ 
10            else  $\text{conflicting} \leftarrow \text{true}$ 
11    if not  $\text{conflicting}$  then return  $a_i$ 
12 return  $\text{null}$ 

```

FIGURE 9.42 The algorithm *Backjumping* is like *Backtracking*, except that the procedure *SelectValueGBJ* tests for consistency with compound labels of increasing size. If at some stage the value being considered is not consistent with the label, it marks the index of the label as a possible culprit. We assume a function *SubLabel* that extracts a compound label of the first k variables from the partial solution being constructed.

Gaschnig's approach to jumping back works only when a dead end is encountered during search, sometimes called a *leaf dead end*. It may be possible that when the algorithm jumps back to the culprit variable, it cannot find another value there as well. This situation is referred to as an *internal dead end*. But here the algorithm is compelled to do chronological backtracking. The reason for this is as follows. Let x_c be the culprit variable that *Backjumping* has jumped back to. Now the fact that the algorithm had earlier proceeded beyond x_c means that it *had* found a

consistent value for x_c . It is only when it has jumped back to x_c that it cannot find *another* consistent value. The reason it cannot apply the same principle while backtracking from x_c , the internal dead end, is that it was consistent with its predecessors (when it went forward) and, therefore, will be consistent with all the prefixes of the predecessors as well. And it cannot find a new culprit by trying the compound label with the remaining values of x_c because it might jump over a variable which might have a value consistent with one of the ignored values of x_c . Hence to be safe, *BackJumping* steps back to the previous variable. The reader should verify that this happens automatically with the value stored in *latest(c)*, and likewise for its predecessors.

In the search tree of Figure 9.34, when the algorithm is unable to choose a value for the variable C , it will jump back to $F = r$, which is the first variable $C = r$ came in conflict with. Before that $C = b$ would have conflicted with $B = b$, but of the two, F is the “latest” variable. Since the algorithm has no other value for F (w conflicts with $G = w$), it steps back one step to the variable C .

An algorithm that is able to jump back from internal dead ends as well is described below. The algorithm *GraphBackjumping* decides which variable is a culprit, based on the topology of the underlying constraint graph. In the process, it tends to be more conservative, and its jumps may not be maximal.

9.8.2 Algorithm *GraphBackjumping*

The reason that a chronological step back may not be useful is that the previous variable may not have imposed any constraint on the current dead-end variable. One simple way to infer that the previous variable did not impose a constraint that resulted in consistency is that it is not connected to the current variable in the constraint graph.

When a search algorithm reaches a dead end at the variable x_{k+1} the algorithm *GraphBackjumping* assumes that the *latest* variable x_l connected to x_{k+1} in the constraint graph is the culprit. Observe that this may not lead to a maximal jump back because the real culprit may in fact be an earlier variable. *GraphBackjumping* does not keep track of values and conservatively assumes that x_l is the culprit. Also, observe that *GraphBackjumping* will be safe only when all constraints explicitly show up in the constraint graph.

If the variable x_l is an internal dead end, the algorithm will jump back to the latest predecessor variable connected to *either* x_{k+1} *or* x_l . The style of backjumping is illustrated in Figure 9.43, where the CSP of Figure 9.33 is attempted with the ordering *CEABFDG*. If the variable G were to be a dead end then the latest connected predecessor is F , but if F too does not have another value then the algorithm will jump back to E which is the next latest predecessor of G .

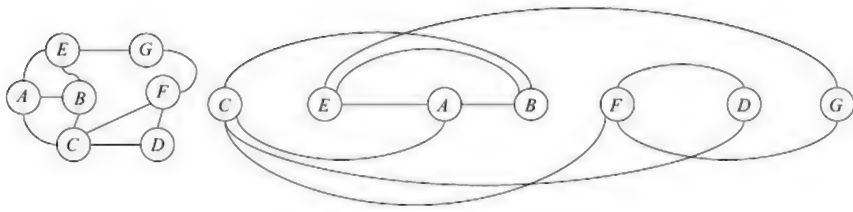


FIGURE 9.43 *GraphBackjumping*. Given the left ordering, if G is a dead end, the algorithm jumps back to F . If that is an internal deadend, it jumps back to E , and then to C if needed.

When the *GraphBackjumping* algorithm retreats from a variable X , it uses a list of predecessors that it may need to jump back to. This list contains not only its own connected predecessors, but also that of any node in the future it is jumping back from. In the above figure, let the algorithm be about to jump back from the node F . If D were to be the leaf dead end that initiated the retreat for some assignment a_F to variables C , E , A , B and F then it would “jump” back to F , and if F had no more values it would now go to C , because that is the latest connected predecessor for both F and D , the node it jumped back from. If on the other hand, G were to be the dead end for an assignment a_D to variables C , E , A , B , F and D , the algorithm would jump back to F , and then to E if it had to retreat further, if F was an internal dead end, finally backtracking to C if needed.

In the algorithm *GraphBackjumping* in Figure 9.44, each variable X maintains a set T_X of target variables that it may have to jump back to. This set is initialized to $Parents(X)$ (the parents of X in the given ordering) when search advances to X .

$$T_X \leftarrow Parents(X)$$

```

GraphBackjumping (X, D, C)
1  A ← ( ( ) )
2  i ← 1
3  Di' ← Di
4  Ti ← Parents(i)
5  while 1 ≤ i ≤ n
6      do ai ← SelectValue(Di', A, C)
7          if ai = null
8              then iCurrent ← i
9                  i ← Latest(Ti)
10                 Ti ← Ti ∪ (TiCurrent ∩ Predecessors(Xi))
11                 A ← JumpTail(A, i, iCurrent)
12             else
13                 A ← Cons(ai, A)
14                 i ← i + 1
15                 if i ≤ n
16                     then Di' ← Di
17                         Ti ← Parents(i)
18  return Tail(Reverse(A))

SelectValue (Di', A, C)
1  while not empty(Di')
2      do ai ← Head(Di')
3          Di' ← Tail(Di')
4          if Consistent(A, xi = ai)
5              then return ai
6  return null

```

FIGURE 9.44 The procedure *GraphBackjumping* is like *Backjumping*, but maintains a list of target variables T_i it can jump back to from X_i . When the time comes, it jumps to the latest variable in T_i at that point. We assume a function *JumpTail*($A, i, iCurrent$) that rolls back the partial solution by an appropriate amount determined by the two indices i and $iCurrent$.

Let the search jump back from some variable Y to X . If there is no value remaining in the domain of X then the algorithm will have to jump back from X . The candidate nodes for jumping back to are the nodes in T_X and the nodes in T_Y that occur before X in the ordering.

$$T_X \leftarrow T_X \cup (T_Y \cap \text{Predecessors}(X))$$

We assume a function *Predecessors*(X) that returns the nodes that are before X in the ordering. Note that T_Y would likewise have been initialized to *Parents*(Y) and augmented, if Y were not a leaf dead end. If Y were to be a leaf dead end then T_Y would be *Parents*(Y).

Assuming that the nodes are visited in the order X_1, X_2, \dots, X_n then the index of the node that *Backjumping* will jump back to from X_i is the latest node in T_i , the target set of X_i . We assume a function *Latest*(T) that returns the index of the latest variable from a set T with respect to the

given ordering.

This making of a more informed leap back from an internal dead end is the way *GraphBackjumping* is different from *Backjumping*. *Backjumping* makes the first jump (from a leaf dead end) that is both safe and maximal, but subsequently moves back one step at a time in the chronological order. *GraphBackjumping* may be somewhat conservative in the first jump from the leaf dead end, but can make longer jumps from internal dead ends too.

Backjumping relies on the actual conflicts of values it has seen. But these apply only to leaf dead ends. *GraphBackjumping* relies on possible conflicts as indicated by the constraint graph. It treats both internal and leaf dead ends in a similar manner. The only additional thing it needs to do for internal dead ends is to keep track of the parents of future variables it has backtracked from. The algorithm *CDBackjumping* or *Conflict Directed Backjumping* relies on the information of actual conflicts, but maintains a target set for jumping back to in the manner of *GraphBacktracking*.

9.8.3 Algorithm CDBackjumping

Conflict Directed Backjumping is an algorithm that is aware of the underlying constraint graph, but determines where to jump back to, based on the actual conflicts that it has recorded. As a result, it can jump back over variables that *GraphBackjumping* is forced to revisit.

The algorithm uses an ordering of the constraints defined as follows. Given an ordering of the variables, a constraint C_i comes earlier than a constraint C_j if the latest variable in its scope S_i which is not in S_j comes earlier in the variable ordering, than the latest variable in S_j not in S_i . For example, if the variables are ordered as (x_1, x_2, \dots, x_n) and the scope of C_1 is $S_1 = \{x_1, x_5, x_7, x_9\}$ and the scope of C_2 is $S_2 = \{x_2, x_5, x_6, x_9\}$ then C_2 comes earlier in the constraint ordering because x_6 comes before x_7 .

Let the *ordered* set of constraints be (C_1, C_2, \dots, C_p) with the corresponding relations (R_1, R_2, \dots, R_p) and scopes (S_1, S_2, \dots, S_p) . When the algorithm has constructed a compound label $a_k = (x_1 = v_1, x_2 = v_2, \dots, x_k = v_k)$ and is looking for a consistent value for the $(k + 1)^{\text{st}}$ variable x_{k+1} , it looks at the values in its domain D_{k+1} one by one. For each value $b \in D_{k+1}$ that is inconsistent with a_k , it identifies the earliest constraint C_j (in the ordering described above) that violates $(a_k, x_{k+1} = b)$. It adds the variables in the scope S_j of that constraint to the set of target variables it may need to jump back to. Like *GraphBackjumping*, this algorithm too accumulates the target variables when it jumps back from a future variable Y to the current variable, and also chooses the latest variable from this set of target variables. The algorithm is depicted in Figure 9.45.

One difference between Gaschnig's *Backjumping* algorithm and *CDBackjumping* is that in the *SelectValue* phase, the former marks the

index of the variable that is the culprit at a leaf dead end, while the latter marks the constraints that participate in a conflict with *any* value of the current variable. As it tries different values $b \in D_{k+1}$, it marks all the (variables in the) constraint that the value $x_{k+1} = b$ was in conflict with. This set of variables called the *Earliest Minimal Conflict (EMC)* set is remembered as the set of variables that conflicted with some value from D_{k+1} . Observe that the *EMC* set is augmented, only if there is an actual conflict which is better than the conservative *GraphBackjumping* that assumes that any connected variable is a culprit. However like *GraphBackjumping*, it remembers the set of target variables, so that when the current variable is an internal dead end it needs to backtrack from, it can jump back safely using this target set augmented by the targets of variables it has jumped back from. The target set T_i for each variable is recorded in the forward phase, and used in the backtracking phase to identify safe variables to jump back to. This is better than Gaschnig's *Backjumping* which had no such memory of target variables and was forced to take only one step back chronologically after the first jump from the leaf dead end.

```

CDBackjumping ( $X, D, C = (C_1, C_2, \dots, C_P)$ )
1   $A \leftarrow (())$ 
2   $i \leftarrow 1$ 
3   $D'_i \leftarrow D_i$ 
4   $T_i \leftarrow ()$ 
5  while  $1 \leq i \leq n$ 
6      do  $a_i \leftarrow \text{SelectValue-CDBJ}(D'_i, A, C, i, T_i)$ 
7          if  $a_i = \text{null}$ 
8              then  $i_{\text{Current}} \leftarrow i$ 
9                   $i \leftarrow \text{Latest}(T_i)$ 
10                  $T_i \leftarrow T_i \cup (T_{i_{\text{Current}}} \cap \text{Predecessors}(X_i))$ 
11                  $A \leftarrow \text{JumpTail}(A, i, i_{\text{Current}})$ 
12          else
13               $A \leftarrow \text{Cons}(a_i, A)$ 
14               $i \leftarrow i + 1$ 
15              if  $i \leq n$ 
16                  then  $D'_i \leftarrow D_i$ 
17                       $T_i \leftarrow ()$ 
18  return Tail(Reverse( $A$ ))

SelectValue-CDBJ ( $D'_i, A, C, i, T_i$ )
1  while not empty( $D'_i$ )
2      do  $a_i \leftarrow \text{Head}(D'_i)$ 
3           $D'_i \leftarrow \text{Tail}(D'_i)$ 
4          conflicting  $\leftarrow \text{false}$ 
5           $k \leftarrow 1$ 
6          while  $k < i$  and not conflicting
7              if Consistent(SubLabel( $A, k$ ),  $x_i = a_i$ )
8                  then
9                       $k \leftarrow k + 1$ 
10                 else
11                     conflicting  $\leftarrow \text{true}$ 
12                      $j \leftarrow \text{EMC}(\text{SubLabel}(A, k), x_i = a_i)$ 
13                      $T_i \leftarrow T_i \cup S_j$ 
14                 if not conflicting then return  $a_i$ 
15  return null

```

FIGURE 9.45 The main procedure of *CDBackjumping* is same as *GraphBackjumping*, maintaining a set T_i of target variables to jump back to. The difference is that here the set T_i is initialized to an empty set, and variables are added in the call to *SelectValue-CDBJ*. The function *SubLabel* extracts the first k values from a compound label. The algorithm works with a set of ordered constraints as described in the text. We assume a function *EMC* that returns the index of the earliest constraint in the ordering that produces the conflict, and variables in the scope of that constraint are added to the target set.

9.9 Discussion

Constraint Satisfaction Problems are an alternative formulation of

problem solving, in which the problem to be solved is represented as a set of variables and constraints that define the combination of values the variables can be assigned in the same solution. Each constraint specifies partial and local information as relations between a subset of variables. Taken together, all the constraints together specify a constraint or relation on all the variables. This *solution relation* specifies the solutions or the set of allowable values for all the variables. The task of solving the CSP is to find this solution relation.

The simplest approach to solving a CSP is to treat it as a search problem, and explore different combinations of assignments till a solution (assignment) is found. The problem with this approach is that the number of combinations is too large. The *explicit* nature of the local information makes it possible to reason about allowable combinations to prune the domains of the variables, throwing away values from domains that will never participate in solutions. This form of reasoning also prunes combinations of values, or relations, for sets of variables, keeping only those that can participate in solutions. Enforcing higher forms of reasoning is also a computationally expensive process, and one has to rely on an appropriate combination of search and reasoning to solve a specific class of problems.

We have studied different combinations of reasoning that can help reduce the load of search, including ordering of variables based on the topology of the graph, combining different levels of lookahead with search, and exploiting some form of information to avoid searching fruitless combinations. The lookahead strategies described in this chapter have introduced the notion of *domain independent heuristics* in search. The heuristics are used to select the variable to instantiate next, as well as the values for the selected variable. The heuristic is computed not by harvesting some domain knowledge as was done in chapter 3, but are computed in a *domain independent manner by exploring the future decisions with a relaxed version of the original harder problem*. We will study this notion of domain independent heuristic functions in the next chapter as well as on advanced planning.

In this chapter, we have confined ourselves to finite CSPs in which the domains of variables are finite sets. Many interesting problems, on the other hand, have infinite and/or real domains. Many of these are well studied in specific forms such as linear programming or integer programming problems. These problems are often associated with the optimization of some objective subject to these specialized constraints. One can also associate optimizing functions with CSPs by introducing what are known as *soft constraints*. Soft constraints are expressed as any other constraints, except that a penalty function is associated with them. In the normal CSP, a solution is an assignment that satisfies all the constraints unequivocally. With soft constraints on the other hand, a solution can be accepted even if the constraints, usually a few are not satisfied. The caveat is that one has to incur a penalty with not satisfying those constraints. Thus problems with soft constraints have costs

associated with solutions. A zero cost solution would be one where all constraints are satisfied. Solutions where some constraints are violated will have a non-zero cost, dependent on which constraints are violated. The problem of solving a CSP with soft constraints thus becomes an optimizing problem in which one has to find a solution that minimizes the penalty or cost.

One particular problem that is of interest is reasoning with time. When reasoning with problems involving time, one may have to deal with a set of variables that represent time. These variables take values from the real domain, and typically there are two approaches to representing time in what are known as *temporal constraint networks*. In one approach, variables are associated with time points, and in the other, with time intervals. In both, one can have constraints between variables, and one may be required to solve such networks. We will not explore such networks in detail, but introduce them briefly in the next chapter in the context of planning with durative actions. The reader interested in greater detail is referred to (Dechter, 2003), (Apt, 2003), and (Rossi et al., 2006).



Exercises

1. Express the 6-queen problem associated with Figure 9.2, with relations in the extension form.
2. The N -queens problem in the chapter uses column labels $\{a, b, c, \dots\}$. An alternate set of labels would be numbers $\{1, 2, \dots, N\}$. Here, the chessboard is viewed as an $N \times N$ array, with the indices (j, k) of each square being the location. The N -queen problem can then be expressed as a collection of N such indices that satisfy certain constraints. Express the constraints in an intensional form. **Hint:** Given two locations $Q_1 = (x_1, y_1)$ and $Q_2 = (x_2, y_2)$, when does Q_1 attack Q_2 ?
3. Consider the CSP $\langle X, D, C \rangle$ where $X = \{x_1, x_2, x_3\}$, $D_1 = D_2 = D_3 = \{1, 2, 3\}$ and there are three constraints $R_{12} = x_1 < x_2$, $R_{23} = x_2 < x_3$ and $R_{31} = x_3 < x_1$. Draw the matching diagram for the CSP. Simulate the algorithm AC-3 on the matching diagram. What can one now observe about the given CSP?
4. Drop the constraints R_{31} from the above problem and repeat the process.
5. A cryptarithmic problem is an arithmetic problem in which the digits have been replaced by some alphabets of a language. The task is to assign distinct digits to the letters in the encrypted problem such that the arithmetic is correct. Express the following cryptarithmic problems as CSPs
 - (a) SEND + MORE = MONEY
 - (b) SIX + SEVEN + SEVEN = TWENTY

- (c) $EAT + THAT = APPLE$
 (d) $SATURN + URANUS = PLANETS$
6. Design a program to generate cryptarithmic problems. Would you begin with a sum and fit numbers to it, or would you start with a word and try and construct the other words meaningfully?
 7. Constructing a crossword puzzle that can be posed as a *CSP* problem. Given an empty crossword grid and sets of words of different lengths, express the crossword problem generator as a *CSP* problem. What kind of constraints will one add if one wants to generate another problem on the same grid, but with different words? Implement a crossword program that generates crosswords from the words taken from the current issue of a college magazine.
 8. Label the edges in the line diagram in Figure 9.46 with labels from Figure 9.13. Do these drawings represent a trihedral object?

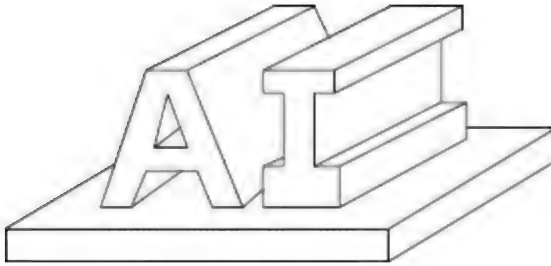


FIGURE 9.46 A line diagram of a solid object.

9. Draw line drawings of objects to illustrate all the different allowed combinations of edge labels given in Figure 9.13.
10. Use the following edge labels to pose the constraint satisfaction problem for scene labelling.
 - (a) (convex $p_1 p_2$)
 - (b) (concave $p_1 p_2$)
 - (c) (occluding $p_1 p_2$)

The last one is to be interpreted as an arrow from p_1 to p_2 . The vertices may be represented as a label and a list of 3 or 4 relevant points,

 - A. (arrow $p_1 p_2 p_3 p_4$)
 - B. (fork $p_1 p_2 p_3 p_4$)
 - C. (tee $p_1 p_2 p_3 p_4$)
 - D. (ell $p_1 p_2 p_3$)
11. Choose an ordering for the *CSP* in Figure 9.21 and show that applying *DPC* and *DAC* makes it backtrack free. Is the same effect achieved if the two procedures are applied in the opposite order?
12. What is the degree of consistency one needs to make the order *EDCBA* for the problem in Figure 9.24, backtrack free?
13. Show that if the constraint graph is a connected graph, then at that point of time when Adaptive Consistency processes the second

variable, there is an edge connecting the second variable to the first variable.

14. Consider the following CSP problem:

$X = \{x_2, x_2, x_3, x_4\}$

$D = \{\{r, b, g\}, \{r, b, s\}, \{b, g\}, \{r, s, g\}\}$

$C = \{C_{ij} \mid i, j \in X\}$ where $S_{ij} = \{x_i, x_j\}$ and $R_{ij} = (x_i \neq x_j)$

Trace the processing done by Adaptive Consistency for the order x_2, x_2, x_3, x_4 . Is the resulting problem backtrack free?

15. Delete any value from the domain of the variable x_4 in Problem 14, and repeat the above process. Delete one more value and try it again.
16. Draw the search tree explored by *Backtracking* for the problem in Figure 9.33 with the ordering *ABCEFDG*.
17. Consider a four variable problem with $X = \{A, B, C, D\}$ and $D_A = \{r\}$, $D_B = \{b, g\}$, $D_C = \{c, m, p\}$, and $D_D = \{o, y, v, i\}$ for some CSP. Compare the sizes of the search trees for orderings *ABCD* and *DCBA*.
18. Draw the search tree explored by the algorithm *Forward Checking* for the problem in Figure 9.33, and compare it with the tree in Figure 9.34.
19. Trace the execution of *Full Lookahead* and draw the matching diagram of the CSP from Figure 9.33 at the point when the algorithm finds the first solution.
20. Show how the algorithm *BackJumping* will backtrack on the search tree of Figure 9.34.

¹ A *future* variable for a search algorithm is a variable that is yet to be assigned a value.

² The missing AC-2 was an algorithm developed by David Waltz for scene labelling, described in Section 9.3. This evolved into AC-3 which has become a much studied algorithm.

³ Since the relations are symmetric, we assume that wherever the relation R_{yz} is available so is R_{zy} without going into implementation details.

⁴ If width is 1, then each node has one parent, and induced width is 1 as well.

⁵ The ordering chosen is not the best possible as observed earlier.

⁶ If the graph is not connected then each connected component is an independent CSP.

⁷ One can also recognize that the task of designing a treat in Chapter 6 is in fact a CSP. This is true of all problems that can be posed as search problems, and CSP is another formulation for a general problem solving method.

⁸ This assumption is made to simplify the algorithm description. In practice, one may pass a pointer to the variable or array.

Advanced Planning Methods

Chapter 10

Planning problems are often described by a Planning Domain Description Language (PDDL) (Fox and Long, 2003; Edelkamp, 2004; Gerevini and Long, 2005). The PDDL allows one to write a set of predicates that describe the domain and operators that describe the actions that are possible. A planning problem is an initial state description and a set of goal predicates that a solution plan is required to satisfy. The task of planning is to find a solution plan.

The simplest plans are sequences of actions, and the most direct approaches to planning are search algorithms to find those action sequences. In Chapter 7, we studied some of the search based planning algorithms for the simplest domains. The simplest domains described in PDDL1.0 have instantaneous deterministic actions making the only changes in a completely observable domain, and the goal predicates specify conditions on the final state. One of the reasons why planning research confined itself to the STRIPS domain is that even for these domains, the basic search algorithms were bogged down by combinatorial explosion.

In this chapter, we look at some of the techniques that increased the length of plans that could be found by an order of magnitude. Included amongst these are methods to adopt a two stage approach in which the problem is transformed into a structure that circumscribes the space of search, and domain-independent, heuristic estimation methods. We also look at the planning in richer domains. In particular, we look at planning with durative actions, and look at approaches for problems with trajectory and soft constraints. We begin with the algorithm *Graphplan* that probably heralded these advanced planning methods.

10.1 GraphPlan

The algorithm *Graphplan* presented by Avrim Blum and Merrick Furst (1995; 1997) takes a very different view of the planning problem. Instead of searching for a solution either in the state space or the plan space (see Chapter 7), it first constructs a structure called a *planning graph* that captures all possible solutions and then proceeds to search for a solution in the planning graph. The *Graphplan* algorithm, as described by Blum and Furst, works in the STRIPS domain. An action is applicable in a state

if its preconditions are true in the state. Negative preconditions cannot be handled¹. The action may have both negative and positive effects. Negative effects delete propositions from the state, and positive effects add propositions.

Figure 10.1 shows the basic difference between the state space for a planning problem and the corresponding planning graph. State space search applies an action to a given state and generates a successor state, as shown on the left. For each action that is applicable, it generates a separate candidate state, which will be a starting point for further exploration. In the figure, we assume two actions a_1 and a_2 that are applicable, with the preconditions linked by edges. The planning graph is a structure, as shown on the right, which *merges* the states produced by the different actions that are applicable. The resulting set of propositions forms a layer, as does the set of actions that resulted in this layer. The planning graph is a layered graph made up of such alternating sets of action layers and proposition layers. This is in contrast to the search tree of states that state space search generates.

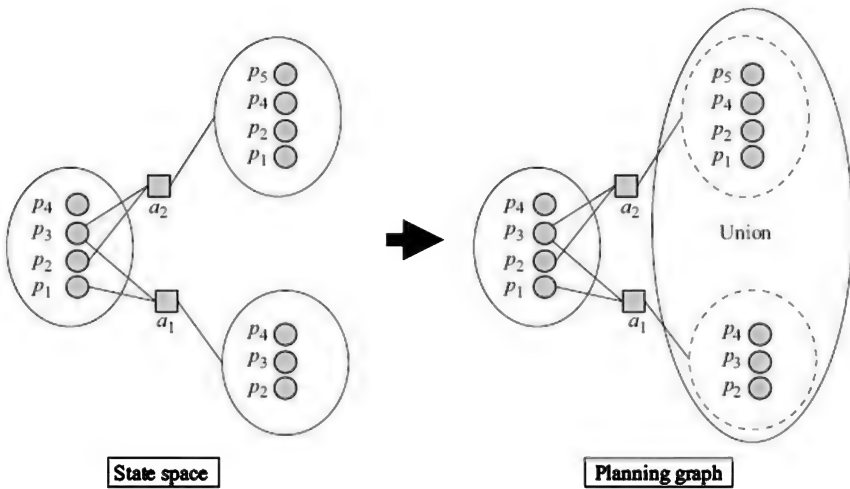


FIGURE 10.1 The planning graph merges the resultant states of all actions that are individually applicable in a given state.

How do we interpret the set of propositions in the proposition layer? Is the union of two (or more) states a state? And what is the semantics of the action layer? Does it mean that the actions in each layer can be executed together? Let us look at a concrete example from the blocks world domain (see Chapter 7). The figure below depicts the action layer and the fact layer for a state containing three blocks. One can see that there are propositions in the new layer that cannot hold together. For example, $On(A, B)$ and $Hold(A)$ ² cannot be true at the same time, and therefore cannot be part of a state. Likewise, $On(A, B)$ and $Clear(B)$, $OnT(C)$ and $Hold(C)$ cannot be a part of a state as well. We can also see

that the two actions in the action layer, $PkUp(C)$ and $UnSt(A, B)$, cannot happen at the same time, given the blocks world assumption of a one armed robot.

Propositions in one layer that cannot be true simultaneously and actions in a layer that cannot happen simultaneously define a binary relation called *mutex* in each layer. The *mutex* relation, which stands for *MUTual EXclusion*, can be shown in the planning graph as edges connecting nodes in a layer. Another question one might ask is about the delete effects of actions. In the above example, the $PkUp(C)$ has a delete effect $OnT(C)$. Should the proposition be allowed in the next layer? The answer is yes, because if the first action in the plan were to be $UnSt(A, B)$, the resulting layer would have $OnT(C)$. This is the Frame Problem (McCarthy & Hayes, 1969) turning up again, and the way it is solved is by introducing a special action called $No-op(P)$ for every proposition P . The $No-op(P)$ action has preconditions $\{P\}$ and effects $\{P\}$. It basically serves to preserve every proposition.

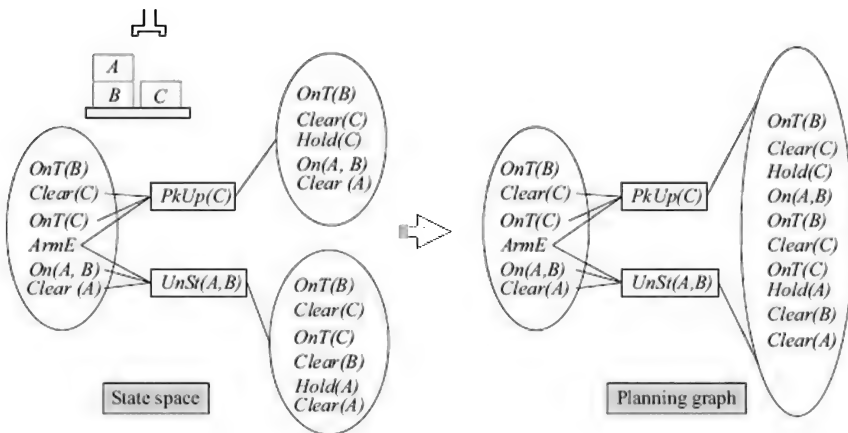


FIGURE 10.2 The proposition layer in the planning graph may contain inconsistent propositions, and the action layer may contain two actions that cannot be done in parallel.

To account for the relation between actions and their effects, we introduce edges between the action in a given layer and each of its effects in the next (proposition) layer. These edges are of two kinds, one for the add effects and the other for the delete effects. A proposition node may have both a positive edge and a negative edge coming in from different actions. Only one, though, can be part of a valid plan.

10.1.1 The Planning Graph

A planning graph, therefore, is a layered graph consisting of alternating layers of proposition and action nodes ($P_0, A_1, P_1, A_2, P_2, A_3, P_3, \dots, A_n, P_n$). For every proposition $p \in P_i$, there is a special action called $No-$

$op(p) \in A_{i+1}$ that carries forward p to P_{i+1} . There are four kinds of edges between nodes.

- A precondition edge $\langle p_i, a_{i+1} \rangle$, linking the precondition p_i to an action a_{i+1} . These are depicted by solid lines in our figures.
- A positive edge $\langle a_i, p_i \rangle$, linking an action a_i to an add effect p_i of a_i . These are also depicted by solid lines in our figures.
- A negative edge $\langle a_i, p_i \rangle$, linking an action a_i to a delete effect p_i of a_i . These are depicted by solid lines with rounded heads in our figures.
- A mutex edge linking two propositions in the same layer, or two actions in the same layer. These are depicted by thick dotted lines in our figures.

The planning graph is made up of the following sets associated with each index i .

- The set of actions A_i in the i^{th} layer.
- The set of propositions P_i in the i^{th} layer.
- The set of positive effect links $PostP_i$ of actions in the i^{th} layer.
- The set of negative effect links $PostN_i$ of actions in the i^{th} layer.
- The set of precondition links $PreP_i$ of actions in the i^{th} layer from P_{i-1} .
- The set of action *mutexes* in the i^{th} layer.
- The set of proposition *mutexes* in the i^{th} layer.

Figure 10.3 illustrates the structure of a planning graph. The initial state of the planning problem defines the layer P_0 . Since P_0 represents a *given state*, there can be no *mutex* edges in this layer. In the graph shown below, there are four actions— a_1 , a_2 , a_3 and a_4 . An action appears in a layer, only if all its preconditions are *nonmutex* in the previous proposition layer. The first two actions a_1 and a_2 appear in the first action layer A_1 , and the other two appear in layers A_2 and A_3 one by one. For every proposition, there is a *No-op* action shown in dashed edges in the figure. Only some of the *mutex* edges have been shown to avoid cluttering up the figure.

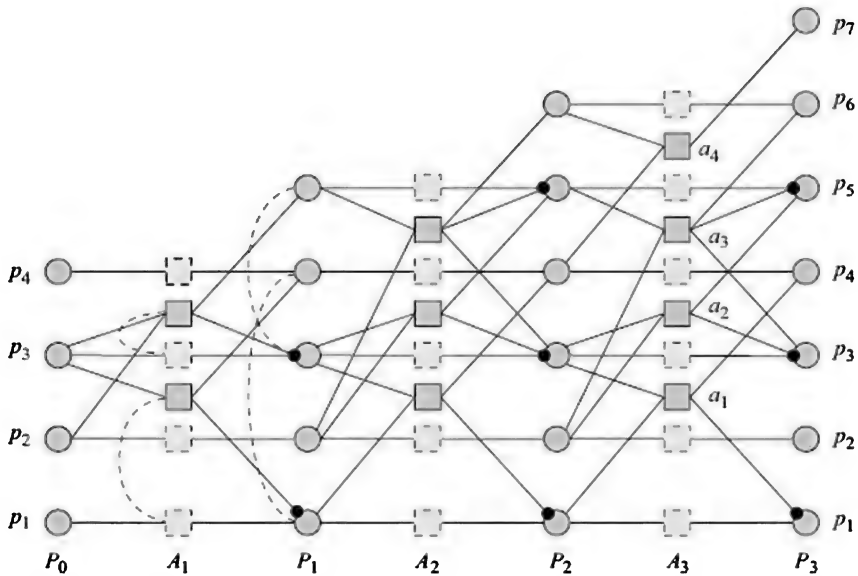


FIGURE 10.3 A planning graph is a layered graph with alternate layers of propositions and actions. The actions with dashed edges are *No-op* actions.

Very often we do not show the *No-op* actions explicitly in the planning graph figures. Instead, we draw edges directly from the proposition in one layer to the proposition in succeeding layers. These are depicted as thick, dashed, grey lines in the figure below. Figure 10.4 depicts a few layers of a planning graph for a simple blocks world problem. Observe that the planning graph does not depend upon the goals to be solved in a specific planning problem.

One of the first observations one can make is that the nodes in the graph grow monotonically with every new layer. This is because every proposition has an associated *No-op* action. Once a proposition appears in a layer, it will always occur in subsequent layers. It is a little less obvious, but when an action appears in an action layer, it occurs in every subsequent layer as well. This is because its preconditions that appeared *nonmutex* in a layer continue to show up *nonmutex* in subsequent layers. This will become clear when we define the mutex relation formally as follows.

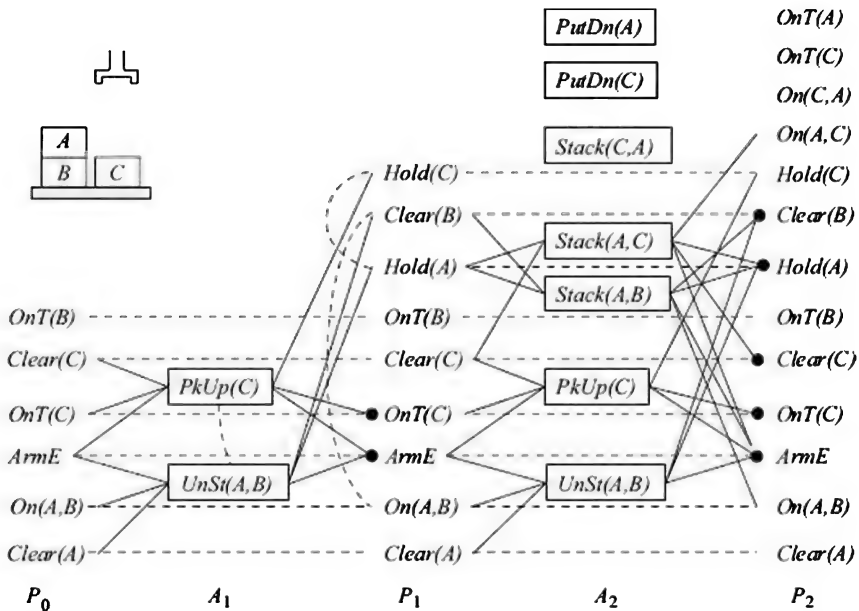


FIGURE 10.4 The planning graph for a simple blocks world problem. Only a few mutex edges are shown. Also, the edges linking some of the new actions in A_2 are not shown.

More interestingly, the proposition layers and action layers in a planning graph stabilize or *level off* after a while. Once the graph has levelled off, no new propositions or actions will appear. However, the number of *mutexes* may still change. In the starting layer, there are no mutexes. But as the graph is built, mutexes appear. However, these mutexes could disappear in future layers. Once they disappear, they can never appear again. Consider the blocks world problem in which two blocks, say A and B , that have to be unstacked from two blocks, say C and D respectively, and are to be put on the table. In the first action layer, there will be two actions, $UnSt(A, C)$ and $UnSt(B, D)$, with effects $Hold(A)$ and $Hold(B)$ in P_1 . These actions and their effects will naturally be mutex since only one block can be picked up by the one armed robot. In the next level A_2 , there are two actions $PtDn(A)$ and $PtDn(B)$, resulting in propositions $OnT(A)$ and $OnT(B)$ in layer P_2 . Again, it is easy to see that these propositions are mutex in P_2 . However, with two more layers of actions and propositions, the two propositions $OnT(A)$ and $OnT(B)$ are *not mutex* in the layer P_4 .

10.1.2 Mutex Relations

The plans produced by *Graphplan* may have actions that occur in the same layer. This means that they could be done in parallel without affecting the outcome. Moreover, if one wants linear plans then these actions could be linearized in any order. The mutex relations identify a

pair of actions or propositions that cannot be present in the same layer in a plan.

Two actions $a \in A_i$ and $b \in A_i$ are mutex if one of the following holds,

1. Competing Needs There exist propositions $p_a \in \text{preconditions}(a)$ and $p_b \in \text{preconditions}(b)$ such that p_a and p_b are mutex (in the preceding layer).

2. Inconsistent Effects There exists a proposition p such that $p \in \text{effects}^+(a)$ and $p \in \text{effects}^-(b)$ or *vice versa*. The semantics of these actions in parallel are not defined. And if they are linearized then the outcome will depend upon the order.

3. Interference There exists a proposition p such that $p \in \text{preconditions}(a)$ and $p \in \text{effects}^-(b)$ or *vice versa*. Then only one linear ordering of the two actions would be feasible.

4. There exists a proposition p such that $p \in \text{preconditions}(a)$ and $p \in \text{effects}^-(a)$ and also $p \in \text{preconditions}(b)$ and $p \in \text{effects}^-(b)$. That is, the proposition is consumed by each action, and hence only one of them can be executed. This condition can be seen as a special case of the condition 3 in which neither linearization is allowed.

Two propositions $p \in P_i$ and $q \in P_i$ are mutex if *all* combinations of actions $\langle a, b \rangle$ such that $a \in A_i$ and $b \in A_i$ and $p \in \text{effects}^+(a)$ and $q \in \text{effects}^+(b)$ are mutex. Observe that if two propositions are not mutex in a layer, they will be not mutex in all subsequent layers because of the *No-op* actions.

10.1.3 Building the Planning Graph

The process of building the planning graph continues till either the goal propositions have appeared nonmutex in the planning graph or the planning graph has levelled off.

1. The procedure to extend the planning graph by one level is described in Figure 10.5. The algorithm *ExtendGraph* takes as input all sets of data for each layer constructed so far, the index i of the layer to be constructed, and the set propositional actions A . The set A may be generated from the start state S and the set of operators O in a pre-processing phase. The algorithm begins by copying the sets for the $(i - 1)^{\text{th}}$ layer, except the *mutexes*. This is because the set of propositions and actions are always carried forward. The algorithm then checks if any new actions are applicable (steps 8, 9). If there are, it adds elements to the different sets (steps 10–17). Then in steps 18 to 23, it computes all the mutex relations that hold at the i^{th} layer in the graph, and

returns the planning graph augmented by one layer in the last step. The applicability of actions in the planning graph needs to check that its preconditions are present *and* that none of the preconditions are mutex with each other. This is done by the procedure *ApplicablePG*(a_k, P_{i-1}, MuP_{i-1}) that takes the action being considered, the previous proposition layer, and the previous proposition mutex layer as input. Note that the test for mutex has both $\langle p_m, p_k \rangle \in MuP_{i-1}$ or $\langle p_k, p_m \rangle \in MuP_{i-1}$ for readability, though strictly speaking, one is enough.

The functions *MutexA* and *MutexP* are left as an exercise for the reader. The first task that algorithm *GraphPlan* takes up is the construction of the planning graph. The algorithm *PlanningGraph* described in Figure 10.6 starts off by initializing the layer zero of the planning graph. This contains only one non-empty set and that is P_0 , which is initialized to the given start state S . The others are initialized to empty lists and the only purpose they serve is as input to the call to *ExtendGraph*. We assume that the procedure is called only if the initial state is not the goal state. The process of extending the graph (lines 11 and 12) continues till any one of the following two conditions is achieved.

1. The newest proposition layer contains all the goal propositions, and there is no mutex relation between any of the goal propositions. This is tested by the procedure *GoalPropExist* in the figure below. Like the *ApplicablePG* procedure, here also the test for mutex has both $\langle p_m, p_k \rangle \in MuP_i$ or $\langle p_k, p_m \rangle \in MuP_i$ for readability, though strictly speaking one is enough.
2. The planning graph has levelled off. This means that for two consecutive levels,

$$(P_{i-1} = P_i \text{ and } MuP_{i-1} = MuP_i)$$

If two consecutive levels have the same set of propositions with the same set of mutex relations between them, it means that no new action can make an appearance. Hence, if the goal propositions are not present in a levelled planning graph, they can never appear, and the problem has no solution.

```

ExtendGraph (i, A, < A0, MuA0, PreE0, PostP0, PostN0, P0, MuP0, ...,
              Ai-1, MuAi-1, PreEi-1, PostPi-1, PostNi-1, Pi-1, MuPi-1>)
1  Ai ← Ai-1
2  Pi ← Pi-1
3  MuAi ← ( )
4  MuPi ← ( )
5  PreEi ← PreEi-1
6  PostPi ← PostPi-1
7  PostNi ← PostNi-1
8  for each ak ∈ (A \ Ai)
9      if ApplicablePG(ak, Pi-1, MuPi-1)
10         then Ai ← cons(ak, Ai)
11             Pi ← append(effects+(ak), Pi)
12             for each p ∈ precond(ak)
13                 PreEi ← cons(<p, ak>, PreEi)
14             for each p ∈ effects+(ak)
15                 PostPi ← cons(<ak, p>, PostPi)
16             for each p ∈ effects-(ak)
17                 PostNi ← cons(<ak, p>, PostNi)
18  for each ak ∈ Ai and am ≠ ak ∈ Ai
19      if MutexA(am, ak, MuPi-1, PreEi, PostPi, PostNi)
20         then MuAi ← cons(<am, ak>, MuAi)
21  for each pk ∈ Pi and pm ≠ pk ∈ Pi
22      if MutexP(pm, pk, PostPi, MuAi)
23         then MuPi ← cons(<pm, pk>, MuPi)
24  return <A0, MuA0, PreE0, PostP0, PostN0, P0, MuP0, ...,
              Ai, MuAi, PreEi, PostPi, PostNi, Pi, MuPi>

ApplicablePG(ak, Pi-1, MuPi-1)
1  Preconds ← precond(ak)
2  if Preconds ⊆ Pi-1
3      then for each pm ∈ Preconds
4           for each pk ∈ Preconds
5               if <pm, pk> ∈ MuPi-1 or <pk, pm> ∈ MuPi-1
6                   then return false
7  else return false
8  return true

```

FIGURE 10.5 The procedure *ExtendGraph* creates the i^{th} action layer and the i^{th} proposition layer. It begins by copying the two preceding layers, and then adds any new applicable actions, and effects, and links to preconditions and effects. After adding all the actions and their effects, it computes afresh the mutex relation for the new action layer and the proposition layer.

The algorithm returns the planning graph along with the level i and the flag *levelled*. If *levelled* = *false* then the planning graph has been extended to the first layer in which the goal propositions exist without any mutex relations between them. The (calling) algorithm now needs to investigate whether a valid plan can be extracted from the planning graph. If *levelled* = *true* then the problem has no solution.

```

PlanningGraph (S, G, A)
1   $P_0 \leftarrow S$ 
2   $A_0 \leftarrow ( )$ 
3   $MuA_0 \leftarrow ( )$ 
4   $MuP_0 \leftarrow ( )$ 
5   $PreE_0 \leftarrow ( )$ 
6   $PostP_0 \leftarrow ( )$ 
7   $PostN_0 \leftarrow ( )$ 
8   $i \leftarrow 0$ 
9  leveled  $\leftarrow$  false
10 while not(GoalPropExist(G,  $P_i$ ,  $MuP_i$ )) or leveled)
11      $i \leftarrow i + 1$ 
12     PlanGraph  $\leftarrow$  ExtendGraph ( $i$ , A,
                                    $\langle A_0, MuA_0, PreE_0, PostP_0, PostN_0, P_0, MuP_0, \dots,$ 
                                    $A_{i-1}, MuA_{i-1}, PreE_{i-1}, PostP_{i-1}, PostN_{i-1}, P_{i-1}, MuP_{i-1} \rangle$ )
13     if ( $P_{i-1} = P_i$  and  $MuP_{i-1} = MuP_i$ ) then leveled  $\leftarrow$  true
14 return  $\langle$ PlanGraph,  $i$ , leveled $\rangle$ 

GoalPropExist(G,  $P_k$ ,  $MuP_k$ )
1  if  $G \subseteq P_k$ 
2  then for each  $p_m \in G$ 
3      for each  $p_k \in G$ 
4          if  $\langle p_m, p_k \rangle \in MuP_k$  or  $\langle p_k, p_m \rangle \in MuP_k$ 
5          then return false
6  else return false
7  return true

```

FIGURE 10.6 The algorithm *PlanningGraph* takes as input the start state S , the goal propositions G , and the set of ground actions A . It begins by extending the graph to layer one. After that, it keeps calling *ExtendGraph* until one of the following conditions become true. One, the goal propositions have appeared nonmutex in the latest layer. Or two, the planning graph has levelled off. The empty sets at level zero have been created only to allow a uniform call to *ExtendGraph*.

Note that the two cases are distinct and there is no possibility of both the tests connected by *or* in step 10 becoming true simultaneously.

The algorithm *Graphplan* is described below. It begins by checking if G is a subset of S . In that case, a plan need not be found. Otherwise, *Graphplan* calls *Planning Graph*. If the procedure *PlanningGraph* returns a planning graph with all goal propositions nonmutex, it is possible, but not necessary, that a valid plan might exist in the graph. We assume a procedure *ExtractPlan*($G, i, PlanGraph$) that either extracts a plan p if there exists one, or returns “nix”.

If the procedure *ExtractPlan* returns “nix” at layer i , there are two possibilities.

The first is that the shortest plan, which may have parallel actions, has more stages than the layers in the planning graph. To investigate this possibility, *Graphplan* extends the planning graph one step at a time, checking whether a plan exists at each stage. Observe that this is a kind of iterative deepening behaviour, and always results in the shortest plans being found. As the planning graph is iteratively extended, it levels off at some stage. Let us call this level n . That is $P_{n-1} = P_n$ and $MuP_{n-1} = MuP_n$.

The second possibility is that a plan does not exist at all, even though

the goal propositions occur without mutex relations amongst them. The question then is: *when should the algorithm stop extending the graph and terminate with failure to find a plan?* Observe that a layer in the planning graph supplies the preconditions for the succeeding layer. Conversely, the layer can be seen as containing the subgoals that need to be solved for the goals in the succeeding layer, and layers beyond that as well. The basic idea in formulating the termination criteria is to identify a layer in the planning graph in which the sets of subgoals are not solvable *and* the sets have stabilized as well. The most obvious³ candidate for this layer is layer n , in which the planning graph has levelled off. If one knew that *all possible* subsets that are candidates for being the subgoal sets at that layer have been considered *and* found to be unsolvable then one can conclude that the planning problem has no solution.

Let us say that the planning graph has gone beyond the level n , and has reached level i at which it has called the *ExtractPlan* procedure. Let S_n^i be the set of sets of subgoal propositions that *ExtractPlan* tried at level n (and failed). Then, if there are two consecutive levels $(t - 1)$ and t such that the size⁴ of S_n^{t-1} is equal to the size of S_n^t then the algorithm can terminate with the output that no plan exists. Observe that the number of sets (at level n) associated with layer $(i + 1)$ would always be greater than sets associated with layer i , because the latter is contained in the former. And the number of such sets that is possible is finite because each set is a subset of P_n . Hence, at some point the number of sets will stop growing. The reader is referred to (Blum and Furst, 1997) for a proof of correctness of this termination criterion.

In the algorithm in Figure 10.7, we have assumed a function *SizeSubgoalSets* that returns the number of different subgoal sets that are possible in the level n when *Graphplan* is attempting to find a plan of i stages. This procedure is dependent upon the *ExtractPlan* procedure. We will describe it with the backward search phase used by Blum and Furst a little later. The algorithm *Graphplan* can be seen to operate in three stages. In the first stage, the initial planning graph is built and if the goal propositions have appeared then it attempts to extract a plan (lines 2–7). If the *ExtractPlan* procedure returns “*nix*”, the algorithm incrementally extends the graph looking for a solution till it levels off (lines 8–15). When it does level off, it marks that level in the variable n , and computes the size of the set of sets of subgoals (lines 12–15). At this stage, this value will be 1 because the subgoal set is the original goal set G here. Finally, in the third phase (lines 16–21), *Graphplan* continues extending the planning graph looking for a plan. This happens till the termination condition is reached in line 21 where it reports failure. If at any time in phases 2 and 3 *ExtractPlan* returns a plan, the algorithm skips both these phases and terminates with the plan.

The plan that algorithm *Graphplan* returns has the following structure,

$$\pi = (\{a_{11}, \dots, a_{1p}\}, \{a_{21}, \dots, a_{2q}\}, \{a_{31}, \dots, a_{3r}\}, \dots, \{a_{m1}, \dots, a_{ms}\})$$

That is, it contains m ordered sets of actions. The first set $\{a_{11}, \dots, a_{1p}\}$ contains actions that can be executed in parallel in stage one, the second set $\{a_{21}, \dots, a_{2q}\}$ in stage two, and so on. If one desired a linear plan then the set for each stage can be linearized in any order.

10.1.4 Extracting the Plan

When the planning graph reaches a level that contains all the goal propositions G in the newest layer with no mutex relations amongst them, it is time to check whether a plan can be extracted from the planning graph. At this stage, we have the goal propositions and the start propositions nonmutex. It remains to be checked whether there exists a plan $\pi = (\{a_{11}, \dots, a_{1p}\}, \{a_{21}, \dots, a_{2q}\}, \{a_{31}, \dots, a_{3r}\}, \dots, \{a_{m1}, \dots, a_{ms}\})$, such that the set of actions at every layer $\{a_{k1}, \dots, a_{kz}\}$ are applicable and non mutex, and that the goal propositions have support from the last set of actions, including the *No-op* actions.

```

Graphplan (start = S, goal = G, actions = A)
1  if  $G \subseteq S$  then return ( )
2  PG  $\leftarrow$  PlanningGraph (S, G, A)
3  PlanGraph  $\leftarrow$  First(PG)
4  i  $\leftarrow$  Second(PG)
5  leveled  $\leftarrow$  Third(PG)
6  if leveled = true then return "no plan exists"
7   $\pi \leftarrow$  ExtractPlan(G, i, PlanGraph)
8  while  $\pi = \text{"nix"}$  and not leveled
9      i  $\leftarrow$  i + 1
10     PlanGraph  $\leftarrow$  ExtendGraph (i, A, PlanGraph)
11      $\pi \leftarrow$  ExtractPlan(G, i, PlanGraph)
12     if ( $P_{i-1} = P_i$  and  $MuP_{i-1} = MuP_i$ )
13         then leveled  $\leftarrow$  true
14         n  $\leftarrow$  i
15          $S_i \leftarrow$  SizeSubgoalSets(i, n, PlanGraph)
16 while  $\pi = \text{"nix"}$ 
17     i  $\leftarrow$  i + 1
18     PlanGraph  $\leftarrow$  ExtendGraph (i, A, PlanGraph)
19      $\pi \leftarrow$  ExtractPlan(G, i, PlanGraph)
20      $S_i \leftarrow$  SizeSubgoalSets(i, n, PlanGraph)
21     if  $S_i = S_{i-1}$  then return "no plan exists"
22 return  $\pi$ 

```

FIGURE 10.7 Algorithm *Graphplan* begins by calling procedure *PlanningGraph* to construct the initial planning graph. *PlanningGraph* returns a triple from the three components which are extracted using the (assumed) functions *First*, *Second* and *Third*. From here on, *Graphplan* calls *ExtractPlan* and extends the planning graph, till either the termination criterion is reached or a plan is found. The function *SizeSubgoalSets* can be computed by inspecting the memory of failed goal sets maintained by *ExtractPlan*.

One way of looking at the plan existence question is to ask whether the goal propositions have support from a nonmutex set of actions. If yes then the combined preconditions of these actions can be viewed as a

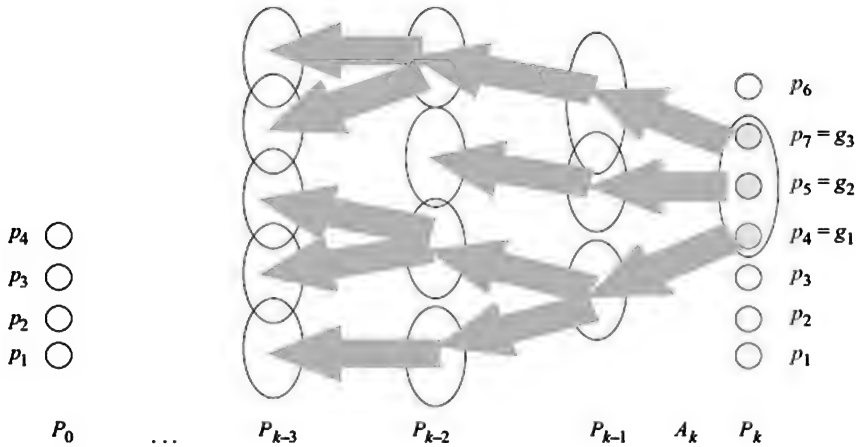


FIGURE 10.9 *Graphplan* does a depth first search, starting from the goal set on the right. A move in the backward search is a regression of the entire goal set to a subgoal set at the preceding layer.

The backward search call (at any recursive level) can terminate either by reaching the level P_0 with success or result in failure. If it is the former then a plan has been found and algorithm *Graphplan* can terminate. If it is the latter, the *ExtractPlan* memoizes the failed goal set. This is a kind of *nogood learning* that can be exploited in future (backward) searches. If a failed goal set is regressed to, in the future the algorithm can immediately backtrack from there without trying to solve it.

The algorithm *ExtractPlan* is described in Figure 10.10. The procedure *RegressGoalSet*, called in line 1, returns *SetOfSubGoalSets* the set of possible subgoal sets along with the actions that regress to them as illustrated in Figure 10.8. Each element of the returned set is a pair $\langle \text{Actions}_i, \text{Subgoals}_{i-1} \rangle$, where Actions_i is the set of nonmutex actions that regress to the nonmutex set of propositions Subgoals_{i-1} in the previous layer. The procedure, which is left as an exercise for the reader, tries all combinations of *relevant* actions (see Chapter 7) that can be chosen in the goal-set layer, such that they achieve the goal set in a nonmutex manner. This set of actions may include *No-op* actions as well. We also assume two functions, *Memoize* and *Memoized*, that manage a memoization memory called *mem*. This memory could be implemented in various ways, for example as list structures or as hash tables. *Memoize(goalSet, i, mem)* adds a set of goals *goalSet* that cannot be solved together at layer *i* to the memory, and *Memoized* checks the memory *mem* to find out whether a goal set is marked as failed at level *i*. Lines 6–16 implement a backtracking based search algorithm that tries all options in *setOfSubgoalSets* one by one. Line 11 checks for the base case in recursion, in which the goal-set layer is P_1 . The actions, which have been tested to be nonmutex in the call to *RegressGoalSet* are returned as the plan. Note that these happen in parallel, or can be linearized in any order. Otherwise, in line 12, a recursive call is made to

ExtractPlan. If the recursive call finds a subplan, it is appended to the actions that generated the subgoal set and returned as a plan. If the subplan was “nix” then *while loop* looks at the next subgoal set. If all goal sets are exhausted, the (calling) goal set is memoized and *Extractplan* returns “nix” (steps 15, 16).

```

ExtractPlan (G, i, PlanGraph)
1  setOfSubgoalSets ← RegressGoalSet(G, i, PlanGraph)
2  setOfSubgoalSets ← FilterMemoized(setOfSubgoalSets, i-1, mem)
3  if empty(setOfSubgoalSets)
4      then mem ← Memoize(G, i, mem)
5      return "nix"
6  while not empty(setOfSubgoalSets)
7      sGoal ← First(setOfSubgoalSets)
8      setOfSubgoalSets ← Rest(setOfSubgoalSets)
9      actions ← First(sGoal)
10     subGoals ← Second(sGoal)
11     if i = 1 then return actions
12     subPlan ← ExtractPlan(subGoals, i-1, PlanGraph)
13     if subPlan ≠ "nix"
14         then return append(subPlan, actions)
15  mem ← Memoize(G, i, mem)
16  return "nix"

FilterMemoized (setOfSubgoalSets, i, mem)
1  for each subGoalSet ∈ setOfSubgoalSets
2      subGoals ← Second(subGoalSet)
3      if Memoized(subGoals, i, mem)
4          then remove subGoalSet from setOfSubgoalSets
5  return setOfSubgoalSets

```

FIGURE 10.10 The procedure *ExtractPlan* does a backward depth first search from a given layer. We assume the function *RegressGoalSets* that regresses to the previous layer. If a call to *ExtractPlan* succeeds, it returns a plan. Otherwise it returns “nix” and also marks the goal set it was called with as ‘failed’ in that layer. We also assume functions *Memoize* which adds to the memory of failed goalsets *mem*, and *Memoized* that checks whether a goalset exists in it.

The algorithm *Graphplan* is guaranteed to find a plan that is shortest in the number of stages in the plan, where a stage consists of one or more *independent* actions.

It has been argued that the planning graph is polynomial in size, in terms of the number of objects, the number of operators with a fixed number of preconditions, the number of add effects and the layer number. And the planning graph can be constructed in polynomial time (Blum and Furst, 1997). Given that the planning problem is known to be hard (Gupta and Nau, 1992), we can infer that most of the work done by *Graphplan* is in the plan extraction phase. Two lines of exploration now present themselves. One is to try and make the plan extraction phase as efficient as possible. Some of the ideas from constraint solvers like forward checking can be applied here. In fact, in the following section, we will explore viewing the plan extraction task as a constraint satisfaction problem. The second is to find other ways to exploit the planning graph in other ways. We will also look at methods to use the planning graph to

generate heuristics to guide state space search.

10.1.5 STAN

One approach to make *Graphplan* more efficient was implemented in the program STAN (*state analysis*) reported by Derek Long and Maria Fox (1999). The reader can observe that the first few lines of the *ExtendGraph* algorithm of Figure 10.5 are devoted to copying information from the previous layer. This incurs a cost both in time and space. Given that both propositions and actions are carried over to succeeding layers, STAN simply keeps one copy of each and keeps track of when the proposition and action was first introduced. Given an initial state and the set of planning operators, the overall set of (ground) actions and propositions is well defined. STAN represents the set of possible ground actions and propositions as bit vectors called the *action spike* and the *fact spike* respectively. The *rank* of an entry in the spike marks its first appearance. A *fact rank* is a consecutive sequence of fact headers of the same rank. Facts are represented using fact headers arranged according to rank, and the headers contain,

1. a name which is the predicate and arguments that comprise the fact itself,
2. an index i , giving the position of the fact in the fact array,
3. a bit mask, which is a fact spike vector in which the i^{th} bit is set and all other bits are unset,
4. a reference identifying its achieving *No-op*,
5. an action spike vector of consumers with bits set for all the actions which use this fact as a precondition, and
6. a fact-level package storing the layer dependent information about that fact.

An action header is likewise made up of,

1. the name of the action,
2. an index i giving the position of the action in the action array,
3. a bit mask which is an action spike vector in which the i^{th} bit is set and all other bits are unset,
4. a flag indicating whether the action is a *No-op*,
5. three fact spike vectors, called *precs*, *adds* and *dels*, and
6. an action level package storing the layer dependent information about that action.

For every action, the *precs* spike vector marks the preconditions of the action. The spike is a fact spike with the preconditions of the action set as 1 and the rest as 0. Thus, testing for applicability of the action is done by a bit-level logical operation. Similarly, the *adds* spike marks the add effects, and *dels* marks the delete effects of the action.

The above information associated with facts and actions is layer independent information. The rank associated with every fact or action

only indicates the first layer in which they appear. The layer-dependent information is stored in a fact-level package and an action-level package.

The fact-level package is an array of pairs, one for each rank in the spike. The pair is made up of,

1. a Fact Mutex Vector (FMV), *mvecs*, which is a fact-spike vector indicating which facts the given fact is mutex with, and
2. an Achievement Vector (AV) which is an action spike indicating which actions add that fact.

The action-level package are made up of,

1. an Action Mutex Vector (AMV), which is an action spike indicating which actions are mutex with the given actions,
2. a list (MA) of actions that are mutex with the given action, and
3. a bit vector *changedActs* to keep track of temporary mutex relations.

Mutexes can also be checked by bit-level logical operations. Actions mutexes may be permanent or temporary. Two actions are permanently mutex if their preconditions, add effects and delete effects interact. Two actions a and b are (permanently) mutex if $((precs(a) \vee adds(a)) \wedge dels(b)) \vee ((precs(b) \vee adds(b)) \wedge dels(a))$ is nonzero. Two actions may be temporarily mutex if some of their preconditions are mutex. Let $\{p_{a1}, \dots, p_{ak}\}$ be the preconditions of action a . Then actions a and b are mutex if $(mvecs(p_{a1}) \vee \dots \vee mvecs(p_{ak})) \wedge precs(a)$ is nonzero. Given that mutexes between actions can only disappear, STAN keeps track of actions becoming nonmutex in the *changedActs* vector to avoid repeated checks for temporary mutexes.

STAN first constructs the spike graph till the “opening layer” where all the goals are present nonmutex. This is analogous to the *PlanningGraph* algorithm of Figure 10.6. After that it alternates between testing for a plan and extending the graph till it has levelled off, or reached the fixed point. If a plan is still not found, STAN leaves the spike graph as it is from here on, since no changes can be made. It does further exploration using a *wavefront*. The idea is that one needs to consider all possible subgoal sets at the fixed point to be able to terminate with failure. STAN maintains a *queue* of goal sets to be considered at the fixed point layer, and a *buffer* in which new goal sets are considered. The new goal sets are the ones that would have been generated if the planning graph had been extended beyond the fixed point. When the graph is (notionally) extended, STAN adds new subgoal sets to the *queue*. When it cannot solve a (new) subgoal set in this *queue*, it propagates forward the goal set into the *buffer*, leading to new goal sets to be added to the *queue*, which it now recedes back to try solving them. This forward and backward movement from the fixed point is characterized by the name *wavefront*. For more details on STAN and on the proofs of soundness, completeness and termination of the wavefront mechanism, the reader is referred to (Long

and Fox, 1999).

10.1.6 Conditional Effects

Another direction of development is to extend the planning graph methods to richer domains. One of the first extensions reported was on ADL set. The Action Description Language (ADL) was a richer alternative to STRIPS domains and was later subsumed in PDDL2.1. The ADL language can be described as (Fox and Long, 2003),

```
adl = :strips + :typing
      + :negative-preconditions
      + :disjunctive-preconditions
      + :equality
      + :quantified-preconditions
      + :conditional-effects
```

Negative preconditions can be handled by introducing a separate proposition to represent a negated one. Consider the proposition *HaveBook(Wren&Martin)*. Consider an action that says that if *HaveBook(Wren&Martin)* is true then one must buy the book. Instead of expressing the negated fact, one can introduce the new proposition *NotHaveBook(Wren&Martin)* and maintain it permanently mutex with *HaveBook(Wren&Martin)*. Then instead of just deleting a proposition in an action, one can *also* add its negation. Observe that this approach requires us to abandon the negation by failure closed world assumption and takes us into an open world formulation. Consider the *Unstack(A, B)* action in the STRIPS domain, which has the proposition *On(A, B)* in the delete list. The closed world assumption says that if *On(A, B)* is not present in the state representation, it is false. That requires the algorithm to scan the representation, if testing for the negation was required. Having explicit negated formulas requires us to maintain one of the two, a proposition or its negation, explicitly. Having both would be inconsistent. Having a disjunction of both could represent uncertainty, as we will see below. One can also think of the add effect *Clear(B)* of the action *Unstack(A, B)* as a kind of expression of the fact $\neg On(A, B)$. And given that only *A* was on *B*, one can in fact assert that nothing is on *B*. This is the way STRIPS handled the frame problem. Thus, $Clear(B) \equiv \neg \exists x On(x, B) \equiv \forall x \neg \exists On(x, B)$. When we have *Clear(B)* as a precondition for the action, say *Stack(C, B)*, we are really testing the quantified precondition it is equivalent to, which also has a negation in it.

Conditional effects do extend the language beyond STRIPS (see (Gazen and Knoblock, 1997), (Koehler et al., 1997), (Anderson et al., 1998)). A conditional effect of an action is an effect that comes into being, only when a specified condition is true. Consider the action of driving a school bus from point *A* to point *B*, *Drive(bus21, a, b)*. Also consider the

action $Board(Person, Bus)$ with effect $In(Person, Bus)$, and the predicate $At(Object, Location)$. What should be the effects of the action $Drive(bus21, a, b)$? Conditional effects allow us to say that when the bus is driven from point A to point B then anyone who had boarded the bus will also be at point B . The $Drive$ operator could be represented as (Koehler et al., 1997),

Action name:	$DriveBus$
Parameters:	Bus (type bus); A, B (type $Location$); X (type $Object$)
Preconditions:	$At(Bus, A)$
Effects:	$Del(At(Bus, A)), Add(At(Bus, B)),$ $\forall x(In(x, Bus) \supset (Del(At(x, A)), Add(At(x, B)))$

The above action could be represented in PDDL as,

```
(:action driveBus
  :parameters (?bus -bus ?from ?to - location)
  :precondition (at ?bus ?from)
  :effect (and (not (at ?bus ?from)) (at ?bus ?to)
              (forall (?x - object)
                (when (in ?x ?bus)
                  (and (not (at ?x ?from)) (at ?x ?to))))))
```

It has been observed that an action with conditional effects can be viewed as a set of actions with different preconditions. To see this, let us first translate the above operator into a quantifier free one for a domain in which there are only two persons who could be in the school bus, Aditi and Jeena. The operator would then look like,

```
(:action driveBus
  :parameters (?bus -bus ?from ?to - location)
  :precondition (at ?bus ?from)
  :effect (and (not (at ?bus ?from)) (at ?bus ?to)
              (when (in aditi ?bus)
                (and (not (at aditi ?from))
                    (at aditi ?to)))
              (when (in jeena ?bus)
                (and (not (at jeena ?from))
                    (at jeena ?to)))))
```

Now this action with conditional effects can be translated into four STRIPS actions, called *aspects* in (Gazen and Knoblock, 1997). The four actions/aspects are,

```

(:action driveBus1
  :parameters (?bus -bus ?from ?to - location)
  :precondition (at ?bus ?from)
  :effect (and (not (at ?bus ?from)) (at ?bus ?to)))

(:action driveBus2
  :parameters (?bus -bus ?from ?to - location)
  :precondition (and (at ?bus ?from) (in aditi ?bus))
  :effect (and (not (at ?bus ?from)) (at ?bus ?to)
              (not (at aditi ?from)) (at aditi ?to))))

(:action driveBus3
  :parameters (?bus -bus ?from ?to - location)
  :precondition (and (at ?bus ?from) (in jeena ?bus))
  :effect (and (not (at ?bus ?from)) (at ?bus ?to)
              (not (at jeena ?from)) (at jeena ?to))))

(:action driveBus4
  :parameters (?bus -bus ?from ?to - location)
  :precondition (and (at ?bus ?from) (in aditi ?bus) (in jeena ?bus))
  :effect (and (not (at ?bus ?from)) (at ?bus ?to)
              (not (at aditi ?from)) (at aditi ?to))
              (not (at jeena ?from)) (at jeena ?to))))

```

However, converting actions into aspects can result in a blow-up in the number of actions the planner has to deal with, and an approach to handle them directly would be desirable. One such approach to extend *Graphplan* to conditional effects was implemented in a planner called *IP²* (Koehler et al., 1997).

The planning graph construction process for *IP²* in the way the new proposition layer is constructed. The operator o with conditional effects can be characterized as,

$prec_o$: the preconditions
 add_o : the unconditional add effects
 del_o : the unconditional delete effects
 $prec_i \dots add_i, del_i$: the i^{th} conditional effect

Like in *Graphplan*, an instance of the operator is added to the j^{th} layer A_j , if the preconditions $prec_o$ are present in the $(j - 1)^{th}$ layer P_{j-1} nonmutex amongst themselves. The unconditional add effects add_o are then added to the j^{th} proposition layer P_j , and if any proposition $p \in del_o$ is present in P_j then a delete link is added between the action and p . After that, for every conditional effect $(prec_i \supset add_i, del_i)$, a proposition $p \in add_i$ is added to the j^{th} layer P if the following conditions hold.

1. $prec_i \subseteq P_{j-1}$
2. All propositions in $prec_i$ are nonmutex with each other in P_{j-1}
3. All propositions in $prec_i$ are nonmutex with all propositions in $prec_o$ in P_{j-1}

When all the goal propositions show up nonmutex in k^{th} proposition layer, *IP²* algorithm embarks upon a plan finding search. Here too, it differs a little from *Graphplan*, in that as it searches backward it also maintains a set of negative goal sets C_j in layers $j < k$ preceding the last layer, along with subgoal sets G_j . These negative goal propositions may be added to prevent undesirable conditional effects. We illustrate their

utility with an example.

Consider the example of the two students, Aditi and Jeena, mentioned above. Let one day the goal be $\{(at\ aditi\ home), (at\ jeena\ school)\}$, perhaps because Aditi was not well. Given that $(at\ aditi\ home)$ is mutex with $(at\ aditi\ school)$, it is imperative that the proposition $(at\ aditi\ school)$ does not appear in the final layer. Otherwise, the goal set will not be nonmutex. Now we need the action $busDrive(bus21, home, school)$ because we need to achieve $(at\ jeena\ school)$ which is a conditional effect in that action. For this conditional effect to be achieved, the condition $(in\ jeena\ bus)$ must be true in the $(k - 1)^{st}$ layer. There is an action $board(jeena, bus)$ that would have achieved $(in\ jeena\ bus)$ when both the bus and Jeena were “at home”. Thus, the plan would have two actions—Jeena boards the bus, and the bus is driven to the school. But how does one ensure that Aditi has not landed up in school as a conditional effect of the same $busDrive(bus21, home, school)$ action as well? IP^2 can observe that the effect $(at\ aditi\ school)$ interferes with the goal proposition $(at\ aditi\ home)$ but it needs the $busDrive(bus21, home, school)$. To ensure that Aditi does not land up in school, it adds a *negative goal* to the $(k-1)^{st}$ layer which says that $(in\ aditi\ bus)$ is part of the negative goal set. The way such a negative goal is handled is that if there is any action that has it as an add effect then that action is not allowed in the backward search phase. Thus, while IP^2 does select the $busDrive(bus21, home, school)$ action, it constructs a plan which does not have the $board(aditi, bus)$ action.

The detailed algorithm is left as an (advanced) exercise for the reader. For hints and a statement on soundness of IP^2 , the reader is referred to (Koehler et al., 1997)

10.1.7 Conformant Graphplan

Very often a planning agent has to deal with uncertain information. This uncertainty can be of different forms. The agent may not know the world completely, or the actions of an agent may not have well-defined effects. We humans face such problems all the time, and have evolved many approaches to address these problems. These include exploiting experience and knowledge based methods, and also probabilistic reasoning. We will explore some of these approaches in later chapters. Here, we look at the problem in the framework of automatic planning.

One approach to planning in the face of uncertain world knowledge is called *contingent planning* (Weld et al., 1998), (Majercik and Littman, 2003), (Albore et al., 2009). The idea here is to do more than find a simple plan. Instead, the aim is to synthesize a plan which can cater to more than one situation. The plans produced in contingent planning incorporate sensing steps that help decide between different courses of actions, all encapsulated in the plan. Bridge players are used to planning the play of their hands in a contingent fashion. For example, a bridge

player might play a few cards in a suit to test how the cards in that suit are divided amongst the opponents. Depending upon the outcome, she chooses one of competing courses of further action. Contingent planning happens in many real world situations. A group of friends planning an evening out may go and try for some movie tickets, and if not available may fall back to visiting a park.

Some well known puzzles are problems of contingent planning. For example, the problem of identifying the defective (heavier or lighter) ball from a set of twelve balls, using a pair of weighing scales. Observe that a move here has different possible effects. For example, you might keep balls 3 and 7 on the left pan, and balls 8 and 9 on the right pan. The balance could go to one of the three states—left heavier, right heavier, or balanced. In contingent planning, one can sense this state. Like most planning problems, one may demand a solution optimal in the number of comparisons one is allowed to make. An interesting problem that has both a contingent solution as well as a conformant one is described in the exercises.

Figure 10.11 illustrates common planning problem faced by residents of the IIT Madras campus. We consider two options available to people if they want to go the Central station to catch a train. They can walk to the MRTS station and take a train. Or they can look around for an autorickshaw and investigate whether the driver is willing⁵ to go to the station, and if one is found hire it. There is another possible obstacle though, in the traffic, for the hiring-auto plan to succeed. A contingent planner may try for an auto before deciding to walk, but a conformant planner would have to commit to the MRTS plan because that succeeds in all *possible worlds*.

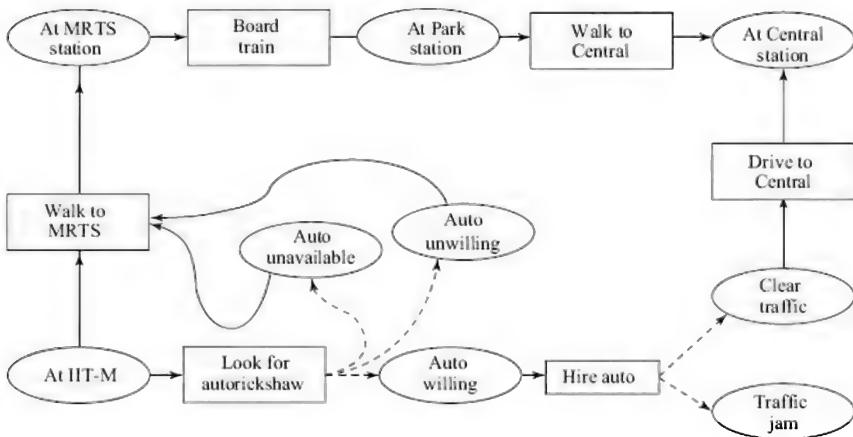


FIGURE 10.11 Reaching a train station. We assume two options are available. The dashed lines represent possible worlds a planner might encounter. A contingent planner could try both options, but a conformant planner would have to commit to a set of actions.

Conformant planning aims at building plans that can cater to different

situations but in which sensing of the state is not allowed. The uncertainty may be in knowing the facts (state) or in the effects of actions (resulting state). David Smith and Daniel Weld show how *Graphplan* can be extended to perform conformant planning in a system call *Conformant Graphplan* (CGP) (Smith and Weld, 1998). CGP represents uncertain knowledge by a disjunction of propositions, or using the exclusive-or. For example, a technician may not be sure whether an instrument has been calibrated or not before using it. Then a conformant plan could be to calibrate it anyway before using it. Smith and Weld approach such uncertainty by constructing planning graphs for each of the possible worlds in which a component of the disjunction (or XOR) might be true, and then finding solutions in each of them.

We illustrate how CGP tackles the bomb disposal problem, first posed by McDermott (McDermott, 1987) as described in (Smith and Weld, 1998). Consider a planning problem in which you have two packages and exactly one of them contains a bomb. This fact is represented by $(In(bomb, package_1) \oplus In(bomb, package_2))$. The exclusive-or entails that there are two possible worlds, one w_1 in which the bomb is in $package_1$ and the other w_2 in which it is in $package_2$. Let there be exactly one toilet that you have access to and an action called *Dunk* defined as follows, which can diffuse the bomb.

```
(:action dunk
  :parameters      (?p - package ?t -toilet)
  :precondition
  :effect          (when (in bomb ?p) (diffused bomb)))
```

The action says that if you dunk a package in the toilet then if the package contains the bomb, the bomb will be diffused. This action has only one aspect with non-empty effects,

```
(:action dunk*
  :parameters      (?p - package ?t -toilet)
  :precondition (in bomb ?p)
  :effect          (diffused bomb)))
```

The action *Dunk** will have two instances—*Dunk*(p1, t)* and *Dunk*(p2, t)*. Each instance will be applicable in the corresponding possible world. CGP constructs a separate planning graph for each possible world, and looks for a solution when the goal set appears nonmutex in all the planning graphs. Figure 10.12 shows the two planning graphs constructed for the possible worlds w_1 and w_2 . It then starts a plan finding exercise moving backwards, level by level, in parallel in each planning graph. If it can find a set of actions in each possible world then it returns the union of the plans found.

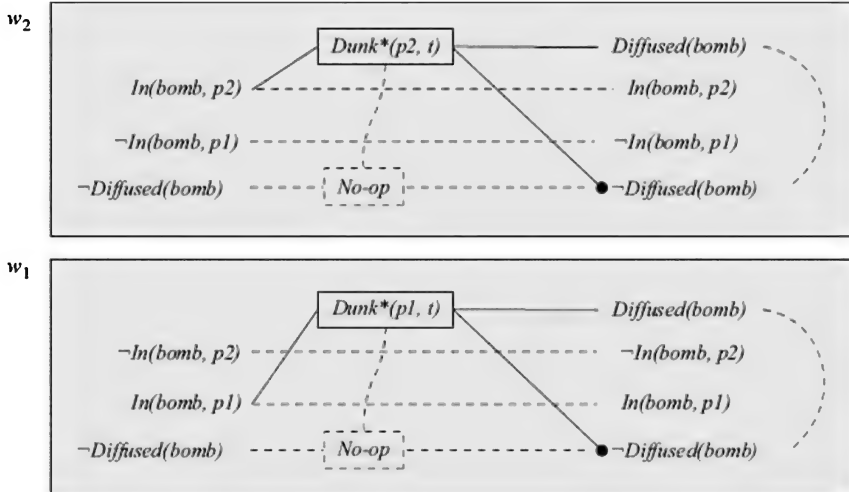


FIGURE 10.12 Conformant Graphplan constructs a planning graph for each possible world.

It finds that the goal *Diffused(bomb)* appears in both the planning graphs at level 1, and further that, the two actions achieve the goal. Thus, it returns a plan of dunking both packages.

The above example does not have interference between the actions across the different planning graphs. To illustrate that case, we augment our dunking action to have an additional effect that it clogs the toilet. The action, *Dunk2*, is given below.

```
(:action dunk2
  :parameters    (?p - package ?t -toilet)
  :precondition  (not (clogged ?t))
  :effect        (and (clogged ?t)
                     (when (in bomb ?p) (diffused bomb))))
```

The modified dunk action requires that the toilet be unclogged to be applicable, and has an effect that it clogs the toilet. Now the two actions of dunking the two packages are individually applicable in the two planning graphs, which are constructed for the two possible worlds. The real world, on the other hand, is one of the two worlds, w_1 and w_2 , but we do not know which. When we try and extract a plan keeping track of interference across worlds, we are unable to select the two dunk actions together, because they are now mutex as they delete the precondition $\sim\text{Clogged}(t)$ required by both, and removed by both. Consequently, we can only choose one of them.

However, both actions appear in both planning graphs, because irrespective of whether the package contains the bomb or not, it has the effect of clogging the toilet. The two aspects of the dunk action are as follows:

```

    (:action dunk2+
      :parameters    (?p - package ?t -toilet)
      :precondition  (and (not (clogged ?t)) (in bomb ?p))
      :effect (and (clogged ?t) (diffused bomb)))
and,
    (:action dunk2-
      :parameters    (?p - package ?t -toilet)
      :precondition  (not (clogged ?t))
      :effect        (clogged ?t))

```

Further, since two possible worlds are in fact complete information variations of a single real world with incomplete information, the two aspects may induce each other in different planning graphs, if the preconditions hold. That is, if we choose the *Dunk2+(p1, t)* action in w_1 , then we have to choose the *Dunk2-(p1, t)* action in w_2 , because we can only choose the actions and not the possible world we are in. Once dunking p_1 is chosen as the action then its relevant aspects have to be chosen in both possible worlds. The situation is depicted in Figure 10.13 below.

In the planning graph for each possible world, we have two dunk actions, one for each package. The specific aspects of these actions that appear depend upon the possible world. But irrespective of which possible world we are in, the aspects of the two actions are mutex, as discussed above. Thus, in each possible world, we can only select one dunk action, and having chosen one, we are then compelled to choose the corresponding induced action in the other world.

We adopt the notation used by Smith and Weld in which $A:w$ stands for action A in world w , and $P:w$ stands for proposition P in world w . Let us say that we choose *Dunk2+(p1, t):w1*. This action results the goal proposition *Diffused(bomb):w1*. But it induces *Dunk2-(p1, t):w2*, which inhibits *Dunk2+(p2, t):w2* being mutex with it. The *No-op:w2 for ~Diffused(bomb):w2* results in the proposition *~Diffused(bomb):w2*. The goal *Diffused(bomb)* appears only in one planning graph and the CPG will continue to extend the planning graphs till it reaches its termination criteria, which in this case is the two graphs levelling off.

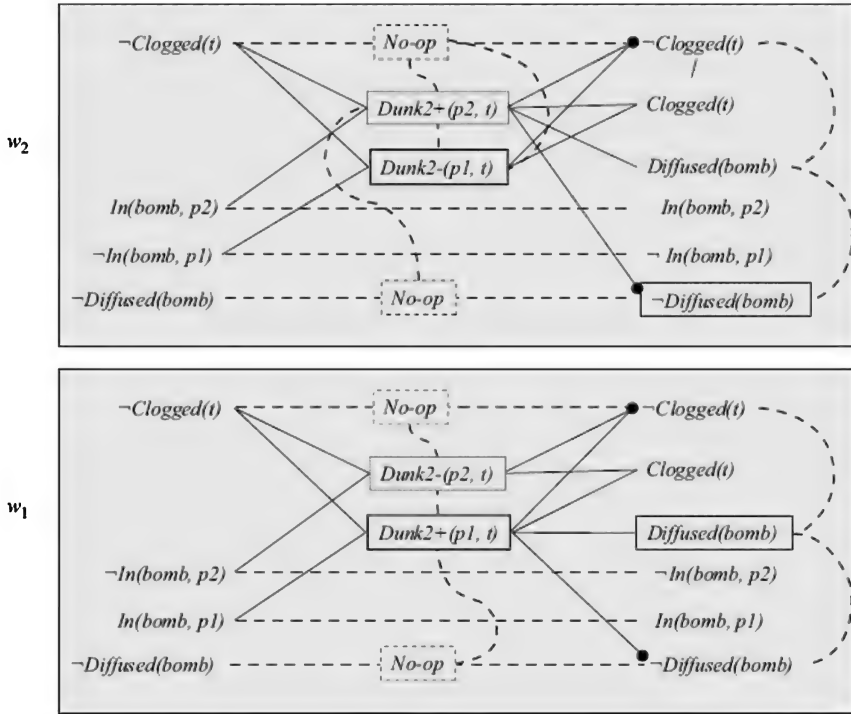


FIGURE 10.13 An action in one planning graph can induce another action in another planning graph. For example $Dunk2+(p1, t)$ in $w1$ induces $Dunk2-(p1, t)$ in $w2$, and vice versa, shown here with thick boxes. The corresponding state of the bomb in the two worlds is indicated by the boxed propositions.

In fact, the two actions $Dunk2+(p1, t):w1$ and $Dunk2+(p2, t):w2$ can be labelled mutex which will avoid the fruitless search that discovers that they cannot happen together. The notion of induced mutex is defined as follows.

Induced Mutex If an action $A:w$ necessarily induces an aspect $A':v$, and if $A':v$ is mutex with an action $B:u$ then $A:w$ is mutex with $B:u$. Note that the possible worlds u and v need not be distinct and the result also applies for $B:v$.

If there were two toilets, t_1 and t_2 , then the CPG would have found the solution of dunking each package in a different toilet. The reader is encouraged to construct the corresponding planning graphs on paper, and verify that there are two distinct conformant plans for diffusing the bomb.

Given an action A that is desirable in some world, one can sometimes prevent an undesirable induced aspect A' in a different possible world. This is done by *confronting* the undesirable aspect by making some precondition of A' false. This is similar to the process in which only desirable aspects are included in IP^2 as described in the previous section.

Conformant GraphPlan can also include uncertainty in the effects of actions. Consider the case when the dunking action may or may not clog the toilet. This could be represented as different possible outcomes in PDDL-like⁶ fashion as follows.

```
(:action dunk3
  :parameters    (?p - package ?t -toilet)
  :precondition  (not (clogged ?t))
  :effect        (or (when(in bomb ?p) (diffused bomb))
                    (and (clogged ?t)
                        (when(in bomb ?p) (diffused bomb)))))
```

This action will have two aspects as before.

```
(:action dunk3+
  :parameters    (?p - package ?t -toilet)
  :precondition  (and (in bomb ?p) (not (clogged ?t)))
  :effect        (or (diffused bomb)
                    (and (clogged ?t) (diffused bomb))))

(:action dunk3-
  :parameters    (?p - package ?t -toilet)
  :precondition  (not (clogged ?t))
  :effect        (or ( )
                    (clogged ?t)))
```

Inclusion of such an aspect in the planning graph in world w_i will result in w_i splitting into two new worlds w_{i1} and w_{i2} in the next layer, containing the two different effects. Observe that action *dunk3-* may not have any effect in some resulting world. If we were to expand the first layer of the planning graph in w_1 then we would include two aspects *Dunk3+*(p_1, t): w_1 and *Dunk3-*(p_2, t): w_1 . These two actions would split the world w_1 into four worlds— w_{11} , w_{12} , w_{13} and w_{14} . In w_{11} , the package p_1 would be dunked, the bomb would be diffused and the toilet clogged; in w_{12} the package p_1 would be dunked, the bomb would be diffused and the toilet would not be clogged; in w_{13} , the package p_2 would be dunked, the bomb would be diffused and the toilet clogged; and in w_{14} , the package p_2 would be dunked, the bomb would not be diffused and the toilet would not be clogged. As one can see, there is going to be an explosion in the number of worlds the planner has to deal with. The solution extraction process, however, remains the same. For more details and for suggestions to keep the number of possible worlds in check, the interested reader is referred to (Smith and Weld, 1998).

More recently, there have been approaches extending and exploiting the planning graph for planning with durative actions. We will explore durative actions later in the chapter.

10.2 Planning as Constraint Satisfaction

The question one asks while extracting a plan from a planning graph is whether the goal propositions are supported by nonmutex actions. Since these actions need to have their preconditions true in the preceding layer, those preconditions become subgoals for which the same question has to be asked. The search process employed by *Graphplan* is similar to the *Backtracking* procedure of solving CSPs. The similarity of extracting plans from planning graphs to solving CSPs extends further. There have been attempts to use 'look forward' approaches borrowed from CSP methods, and the memoization employed by *Graphplan* is similar to *nogood* learning methods (see (Dechter, 2003)) employed in solving CSPs. The mutex relations in planning layers are like arc consistency requirements in CSPs. In fact, it has been observed that if higher order mutual exclusions constraints (or nogoods) were maintained then the backward search process would be bound to succeed. This is equivalent to the situation in solving CSPs when an appropriate level of i-consistency makes the search backtrack free.

The plan extraction task can be posed as a CSP problem. In GP-CSP developed by Do and Kambhampati, the variables are the goal and subgoal propositions at different layers (Kambhampati et al., 1997), (Kambhampati, 2000) (Do and Kambhampati, 2000; 2001). The domains of these variables contain the set of actions that produce these propositions. There are two approaches to solving a planning graph as a CSP.

The first uses the notion of a Dynamic CSP (DCSP) (Mittal and Falkenhainer, 1990), (Dechter, 2003) in which a unary constraint called *active* identifies the propositions that participate in the solution. In a DCSP, only a subset of the variables are active, and the task is to find a satisfying assignment for those variables. In solving the planning graph as DCSP, the variables or goal propositions that are active is determined dynamically in the solution finding process. Initially, the goal propositions in the final layer are marked active, and when their values or actions are chosen, the corresponding preconditions are also marked active.

Let us say that the goal in the planning problem depicted in Figure 10.4 is $\{On(A, C), OnT(C)\}$. The variables of the CSP would be all the propositions that occur in the layers P_1 and P_2 in the figure, and their domains the actions in the planning graph that produce them. In the notation below, the subscript refers to the layer the proposition occurs in. The propositions in layer P_0 can implicitly be assumed to have value *true*, or they can be considered to have a value *Init* for an initial action that generates the starting state (in the style of plan space planning—see Chapter 7).

Variables $Clear_2(A), On_2(A, B), ArmE_2, OnT_2(C), Clear_2(C), OnT_2(B),$
 $Hold_2(A), Clear_2(B), Hold_2(C), On_2(A, C), On_2(C, A),$
 $OnT_2(A), Clear_1(A), On_1(A, B), ArmE_1, OnT_1(C), Clear_1(C),$
 $OnT_1(B), Hold_1(A), Clear_1(B), Hold_1(C), Clear_0(A), On_0(A, B),$

$ArmE_0, OnT_0(C), Clear_0(C), OnT_0(B).$

Domains $Clear_2(A) : \{No-op\},$
 $On_2(A, B) : \{No-op\},$
 $ArmE_2 : \{No-op, Stack(A, B), Stack(A, C), Stack(C, A),$
 $PutDn(C), PutDn(A)\},$
 $OnT_2(C) : \{No-op, PtDn(C)\},$
 $Clear_2(C) : \{No-op\},$
 $OnT_2(B) : \{No-op\},$
 $Hold_2(A) : \{No-op, UnSt(A, B)\},$
 $Clear_2(B) : \{No-op, UnSt(A, B)\},$
 $Hold_2(C) : \{No-op, PkUp(C)\},$
 $On_2(A, C) : \{Stack(A, C)\},$
 $On_2(C, A) : \{Stack(C, A)\},$
 $OnT_2(A) : \{PtDn(A)\},$
 $Clear_1(A) : \{No-op\},$
 $On_1(A, B) : \{No-op\},$
 $ArmE_1 : \{No-op\},$
 $OnT_1(C) : \{No-op\},$
 $Clear_1(C) : \{No-op\},$
 $OnT_1(B) : \{No-op\},$
 $Hold_1(A) : \{UnSt(A, B)\},$
 $Clear_1(B) : \{UnSt(A, B)\},$
 $Hold_1(C) : \{PkUp(C)\}$

$Clear_0(A), On_0(A, B), ArmE_0, OnT_0(C), Clear_0(C), OnT_0(B).$

The mutex relations are implemented as constraints. Since actions are not variables but values of variables, action mutexes are modelled as constraints between the corresponding proposition variables. If two actions in a layer are mutex then two propositions in that layer cannot simultaneously take those actions as values. For example, the actions $PkUp(C)$ and $UnSt(A, B)$ are mutex in action layer A_1 in the figure. This means that $Hold_1(C) = PkUp(C)$ and $Hold_1(A) = UnSt(A, B)$ cannot hold together, and likewise $Hold_1(C) = PkUp(C)$ and $Clear_1(B) = UnSt(A, B)$. This is modelled in GP-CSP as,

$$Hold_1(C) = PkUp(C) \supset Hold_1(A) \neq UnSt(A, B)^7$$

and,

$$Hold_1(C) = PkUp(C) \supset Clear_1(B) \neq UnSt(A, B)$$

The remaining action mutexes in the problem of 10.4 are left as an exercise for the reader.

The dynamic CSP starts of with an initial set of constraints derived from the goal propositions. These constraints say that the goal propositions are *active*.

Initial State Active $\{On_2(A, C), OnT_2(C)\}$

The goal propositions are regressed to subgoal propositions by a set of *activity constraints*. Given an action (value), the activity constraint relates the active proposition to its preconditions. The activity constraints are of the form,

$$p_i = a_i \supset Active\{precond(a)_{1-1}\}$$

For our example, the activity constraints are the following:

Activity constraints

$$On_2(A, C) = Stack(A, C) \supset Active \{Hold_1(A), Clear_1(C)\}$$

$$OnT_2(C) = No-op \supset Active \{ OnT_1(C)\}$$

$$OnT_2(C) = PtDn(C) \supset Active \{Hold_1(C)\}$$

The proposition mutexes are encoded in a straightforward manner. If two propositions p and q are mutex in a layer i , they are expressed as,

$$Active(q_i) \supset \neg(Active(p_i)) \text{ or } \neg(Active(q_i) \wedge Active(p_i))$$

A simple procedure for solving a DCSP will need to start with assigning values to active variables, and then proceed in a backtracking like manner, considering one new active variable at a time. In the process, new variables may be activated. When the algorithm is applied to the DCSP generated by encoding the planning graph, this process mimics the backward depth-first procedure adopted by *Graphplan*.

Another approach would be to compile the DCSP into a standard CSP. One could then use the different approaches used to solve CSPs (see Chapter 8). Solving this CSP would not be constrained by the backward direction ordering of variable imposed by DCSP. More importantly, separating the encoding (into a CSP) phase from the solving phase means that one can use state of the art CSP solvers.

A DCSP marks certain variables (goal propositions) as active. Only the active propositions are assigned values (actions) and participate in the plan. The distinction between active propositions and inactive ones can be marked by adding another value to the domain of that proposition. This value \perp , read as *false* or *bottom*, signifies that the proposition is inactive. In other words, a proposition P is active, if and only if $P \neq \perp$.

With this new value added to the domain, all the propositions in all the layers are treated equally and participate in the solution finding process together. In the solution, only the active variables (propositions) are assigned values (producing actions) and the rest are assigned \perp .

Every $Active(P)$ statement in the DCSP is replaced by $P \neq \perp$. The constraints introduced in the DCSP are now written as follows,

The activity constraint activates the preconditions of the action a chosen for a goal proposition p at level i .

$$P_i = a_i \supset Active\{precond(a)_{1..1}\}$$

Let $precond(a) = \{p^1, p^2, \dots, p^k\}$. Then the above activity constraint is written as,

$$p_i = a_i \supset p^1_{i-1} \neq \perp \wedge p^2_{i-1} \neq \perp \wedge \dots \wedge p^k_{i-1} \neq \perp$$

If $G = \{g^1, g^2, \dots, g^n\}$ is the goal set in the final layer κ of the planning graph, the corresponding constraints in the CSP are,

$$g^1_\kappa \neq \perp \wedge g^2_\kappa \neq \perp \wedge \dots \wedge g^n_\kappa \neq \perp$$

The proposition mutex $\neg(Active(q_i) \wedge Active(p_j))$ is encoded as,

$$\neg(q_i \neq \perp \wedge p_j \neq \perp) \text{ or } q_i = \perp \vee p_j = \perp$$

The procedure described above allows us to encode a planning graph into a CSP. But is it necessary to first construct the planning graph and then encode it into a CSP? The answer is no, and it was shown by Peter van Beek and Xinguang Chen, in a system called *CPlan*, that one can directly encode a planning problem with a bound on the number of layers as a CSP (van Beek and Chen, 1999). We look at a way of encoding a planning problem using the state-variable representation as described in (Ghallab et al., 2004).

10.2.1 CSP from State-Variable Representation

We look at an approach to encode a planning problem directly into a CSP using the same example. The state-variable representation uses functions on variables instead of predicates. For example, instead of using the predicate $On(x, y)$, we use a function $On(x)$ and indicate that (block) A is on B by the statement, $On(A) = B$. $On(A)$ is called a state variable, and it can take a value of type *Block*. Let S be of type state. The initial state as shown in Figure 10.4 is represented as,

$$\{On(A, S_0) = B, On(B, S_0) = table, On(C, S_0) = nil, OnT(A, S_0) = 0, OnT(B, S_0) = 0, OnT(C, S_0) = 1, Clear(A, S_0) = 1, Clear(C, S_0) = 1, Clear(B, S_0) = 0, Holding(arm, S_0) = nil\}$$

The second argument in the above functions is a state identifier. If the

planning system reasons only with the current state, the state parameter can be left implicit. We could then describe the state S_0 as,

$$S_0 = \{On(A) = B, On(B) = nil, On(C) = nil, OnT(A) = 0, OnT(B) = 1, OnT(O) = 1, Clear(A) = 1, Clear(C) = 1, Clear(B) = 0, Holding(arm) = nil\}$$

The state is described by a set of state variables— $On(A)$, $On(B)$, $On(C)$, $OnT(A)$, $OnT(B)$, $OnT(C)$, $Clear(A)$, $Clear(B)$, $Clear(C)$, and $Holding(arm)$. These variables can take values from their respective domains. For example, $D_{On(A)} = \{B, C, nil\}$, $D_{On(B)} = \{B, C, nil\}$, $D_{Clear(A)} = \{0, 1\}$, and $D_{Holding(arm)} = \{A, B, C, nil\}$. The values for these variables are either domain objects, which may be typed—for example $Holding(arm)$ can take a value of type *Block* or *nil*—or of a different sort, for example $Clear(A)$ can take a 0 or 1 value. One can observe that this representation lends itself naturally to posing the planning problem as a CSP. Also observe that we have specified values of variables like $Clear(B, S_0) = 0$ in the spirit of completely specifying the start state. In the classical representation used by STRIPS, only the propositions that are *true* needed to be expressed.

The goal can in turn be represented by a set of constraints. Let us say the goal (again) is simply that *A* should be on *C*, and *C* should be on the table. Assuming that we are looking for a plan of maximum k steps, we expect the following sequence of states to occur— S_0, S_1, \dots, S_k . Also we expect the final state to satisfy the goal constraints,

$$\{On(A, S_k) = C, OnT(C) = 1\}$$

The state transitions are produced by instances of operators. The operators can be defined as before, using the preconditions and effects. For example, we use $UnSt(x, y)$ to represent the *Unstack* action as used by STRIPS in the blocks world.

$$\begin{array}{ll} UnSt(x, y) & \\ \text{preconditions:} & On(x) = y, Holding(arm) = nil, Clear(x) = 1 \\ \text{effects:} & Holding(arm) \leftarrow x, Clear(y) \leftarrow 1 \end{array}$$

and likewise, the *Stack* action as,

$$\begin{array}{ll} Stack(x, y) & \\ \text{preconditions:} & Holding(arm) = x, Clear(y) = 1 \\ \text{effects:} & On(x) \leftarrow y, Holding(arm) \leftarrow nil, Clear(y) \leftarrow 0 \end{array}$$

The *PkUp* and *PtDn* (putdown) action can be similarly defined, and are left as an exercise.

Given the start state and the goal state, the task of planning is to find a sequence of actions that will result in the goal states. Let us assume that the plan is a linear plan of bounded length k . This means we are

looking for a sequence of actions $(a_1, a_2, \dots, a_k)^8$. Each element of this sequence is an *action variable*. The domain of action variables contains the names of actions that are possible, including the *No-op*. Thus, for every action a , the domain D_a is defined as follows:

$$D_a = \{UnSt(A, B), UnSt(A, C), UnSt(C, B), UnSt(C, B), UnSt(B, A), UnSt(B, C), Stack(A, B), Stack(A, C), Stack(C, B), Stack(C, B), Stack(B, A), Stack(B, C), PkUp(A), PkUp(B), PkUp(C), PtDn(A), PtDn(B), PtDn(C), No-op\}$$

The *No-op* action is needed in cases where the plan found has a length smaller than k . In the CSP, the task is to find an appropriate value from the above domain for each of the action variables a_i , $1 \leq i \leq k$. The values for the action variables are determined by,

1. The start state
2. The goal constraints
3. The relation between actions and their preconditions
4. The relation between actions and their effects

The action $act(u, \dots, z)$ chosen as a value for the i^{th} action variable should be such that all its preconditions satisfy the state S_{i-1} and all its effects satisfy the state S_i .

Let the action $act(u, \dots, z)$ have m preconditions. For every precondition pre_j , $1 \leq j \leq m$, of $act(u, \dots, z)$ of the form $x_j(arg_1, \dots, arg_p) = v$ the precondition must be satisfied by the preceding state S_{i-1} . This results in binary constraints of the form,

$$\langle a_i = act(u, \dots, z), x_j(arg_1, \dots, arg_p, S_{i-1}) = v \rangle \text{ for } 1 < j \leq m, \text{ for } 1 \leq i \leq k$$

Let the action $act(u, \dots, z)$ have n effects. For every effect e_j , $1 \leq j \leq n$, of $act(u, \dots, z)$ of the form $X_j(arg_1, \dots, arg_p) \leftarrow v$, the constraint $X_j(arg_1, \dots, arg_p) = v$ must be satisfied by the succeeding state S_i .

$$\langle a_i = act(u, \dots, z), x_j(arg_1, \dots, arg_p, S_i) = v \rangle \text{ for } 1 \leq j \leq n, \text{ for } 1 \leq i \leq k$$

Consider the action $UnSt(x, y)$ described above. It has three preconditions, $On(x) = y$, $Holding(arm) = nil$, $Clear(x) = 1$, and two effects $-Holding(arm) \leftarrow x$ and $Clear(y) \leftarrow 1$. We get the set of constraints that are all possible instances of the following. The constraints are defined for each action variable a_i , $1 \leq i \leq k$.

$$\begin{array}{ll}
\langle a_i = \text{UnSt}(x, y), \text{On}(x, S_{i-1}) = y \rangle & \text{for } 1 \leq i \leq k \\
\langle a_i = \text{UnSt}(x, y), \text{Holding}(\text{arm}, S_{i-1}) = \text{nil} \rangle & \text{for } 1 \leq i \leq k \\
\langle a_i = \text{UnSt}(x, y), \text{Clear}(x, S_{i-1}) = I \rangle & \text{for } 1 \leq i \leq k \\
\langle a_i = \text{UnSt}(x, y), \text{Holding}(\text{arm}, S_i) = x \rangle & \text{for } 1 \leq i \leq k \\
\langle a_i = \text{UnSt}(x, y), \text{Clear}(y, S_i) = I \rangle & \text{for } 1 \leq i \leq k
\end{array}$$

The *instances* of the above constraints will have the parameters x and y replaced by A , B and C wherever possible.

However, these constraints are not enough. If we want *every* solution of the resulting CSP to be a valid plan, we must also assert that all the effects of actions have been taken into account. In other words, we should add constraints to circumscribe the effects of actions. We should say that if any state variable is not affected by the action a_i then it remains unchanged in the succeeding layer. These constraints are derived from the *Frame* axioms, which say that the only change that happens is that due to the known actions.

Let $a_i = \text{act}(u, \dots, z)$ be the action chosen for the i^{th} step. Let x be a state variable not affected by this action. That is, it does not figure in the effects of the action. Then the *Frame* axiom states that the value of x must be same before and after the action. The *Frame* axioms are expressed as ternary constraints of the form,

$$\langle a_i = \text{act}(u, \dots, z), x(\text{arg}_1, \dots, \text{arg}_p, S_{i-1}) = v, x(\text{arg}_1, \dots, \text{arg}_p, S_i) = v \rangle$$

for $1 \leq i \leq k$

Let us look at the example of the action $\text{Stack}(A, C)$. The effects of this action are $\text{On}(A) \leftarrow C$, $\text{Holding}(\text{arm}) \leftarrow \text{nil}$, and $\text{Clear}(C) \leftarrow 0$. That is, it affects the variables $\text{On}(A)$, $\text{Holding}(\text{arm})$, and $\text{Clear}(C)$. For every other variable, we must introduce constraints of the form (shown for $\text{Clear}(B)$ and $\text{On}(C)$ only),

$$\langle a_i = \text{Stack}(A, C), \text{Clear}(B, S_{i-1}) = v_{CB}, \text{Clear}(B, S_i) = v_{CB} \rangle \text{for } 1 \leq i \leq k$$

$$\langle a_i = \text{Stack}(A, C), \text{On}(C, S_{i-1}) = v_{OC}, \text{Clear}(C, S_i) = v_{OC} \rangle \text{for } 1 \leq i \leq k$$

Here, v_{CB} and v_{OC} are variables. The only constraint that these variables must satisfy is that they must have the same value in S_{i-1} and S_i . The *Frame* axioms ensure that the only state changes that are discovered by solving the CSP are those that are the effects of the actions in the plan. Figure 10.14 below shows the matching diagram (see Chapter 8) for one action variable a_i and only for one of its values $\text{UnSt}(A, B)$. We also illustrate the ternary constraint, due to the *Frame* axiom for one variable $\text{On}(B)$.

Given this snippet of the matching diagram, one can imagine how the full constraint graph or the full matching diagram will look like. One can think of the variables arranged in layers like in the planning graph. The unary constraints imposed by the start state and the goal propositions can be used to enforce node consistency in the corresponding layers of state variables. The reduction in their domains will be propagated by arc consistency via the constraints linking state variables to actions, and onto other state variables, and so on. In fact, any of the CSP solving

algorithms can profitably be employed to find a solution that will contain the plan in the values of the action sequence (a_1, a_2, \dots, a_k) . There may be *No-op* actions in the plan found, and they can be deleted to yield a shorter plan. One can now implement an iterative deepening like algorithm that will look for plans of increasing length bounds.

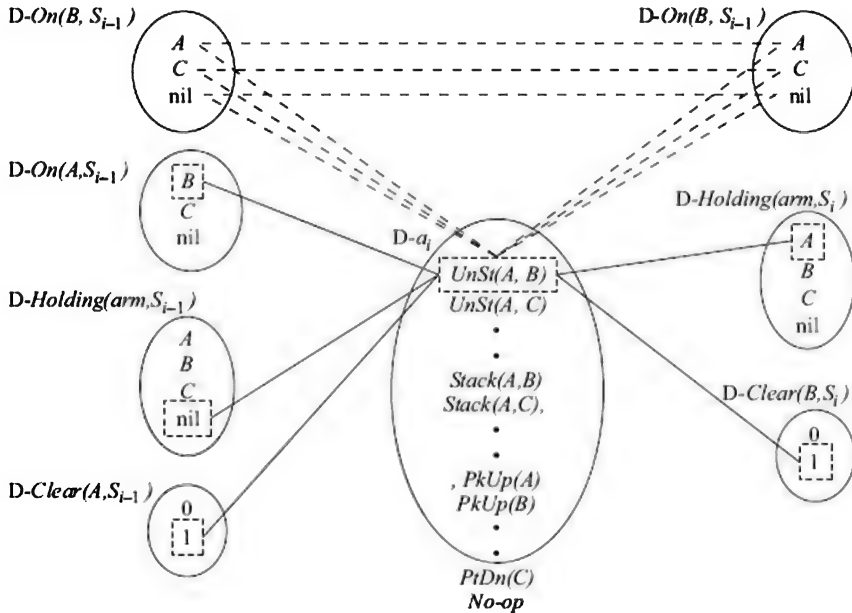


FIGURE 10.14 The matching diagram for constraints associated with the action a_i in the i^{th} stage. The illustration shows the case when the action in question is $UnSt(A, B)$. Other actions will have similar links. The thick dashed triangles represent the ternary constraints associated with the *Frame* axioms, shown only for the variable $On(B)$.

10.3 Planning as Satisfiability

Where the state-variable representation was conducive to posing planning as a CSP, the classical representation can similarly be used to pose planning as a satisfiability (SAT) problem. This too results in a two stage process of solving a planning problem. The first stage involves encoding a planning problem as SAT and the second stage then uses well studied techniques to solve the SAT problem. A well known series of algorithms under the name SATPLAN by Henry Kautz and Bart Selman and David McAllester is based on this idea (Kautz and Selman, 1992; 1996; 1998), (Kautz et al., 1996),

In the classical representation, the state is represented by a set of true propositions. If we treat each possible proposition as a Boolean variable, then we have the basis of encoding the planning problem as a satisfiability problem. Encoding planning as SAT is similar to encoding it into a CSP. Instead of constraints, the problem is expressed as a formula

in propositional logic, and looks for a satisfying assignment to the propositional variables. The propositional variables, for example $On(A, B)$, are called *fluents*, predicates that can change their value over time.⁹ We will represent fluents by the symbols f, f_1, f_2, f_3 , and so on, depending upon the situation.

10.3.1 Linear Encodings

The basic idea in the so called ‘linear’ encoding of planning into satisfiability is to add a time parameter to the fluents and express the relation between actions, preconditions and effects as well as the *Frame* axioms (Kautz and Selman, 1992). The SAT formula to be solved is composed of a set of clauses as follows,

1. Fluents from the initial state S_0 . If $f \in S_0$ then f_0 is a clause, else $\neg f_0$ is a clause. In the planning example from Figure 10.4, we have the fluents $On(A, B, t_0)$, $Clear(A, t_0)$, $\neg Clear(B, t_0)$ for example.
2. Fluents from the goal state. Assuming that we need a plan of at most k steps, we add a fluent f_k for every goal proposition $f \in G$. Observe that the goal G is usually incompletely specified. We only add the known fluents to the SAT encoding.
3. An action implies its preconditions. If an action holds (is true) at time t then its preconditions must hold at time $(t - 1)$ ¹⁰. This is expressed as a clause $(a_t \supset precond(a)_{t-1})$ where $precond(a)$ is the conjunction of the preconditions of action a . For example we have, $(UnSt(A, B, t) \supset (On(A, B, t-1) \wedge ArmEmpty(t-1))^{11} \wedge Clear(A, t-1))$.
4. An action implies its effects, at time t . This is expressed as a clause $(a_t \supset effects(a)_t)$ where $effects(a)$ is a conjunction of the positive and negative effects of action a . For the unstack action we have, $(UnSt(A, B, t) \supset (\neg On(A, B, t) \wedge \neg ArmEmpty(t) \wedge Holding(A, t) \wedge Clear(B, t)))$.
5. Classical *Frame* axioms. If the action a does not affect a fluent f , then f remains unchanged after the action. $(f_{t-1} \wedge a_t) \supset f_t$. For example $((Clear(C, t-1) \wedge UnSt(A, B, t)) \supset Clear(C, t))$. An alternative formulation of the *Frame* axioms, known as *explanatory frame axioms*, has two clauses for each fluent, which say that if the fluent has changed then one of the actions that causes that change must have happened (Haas, 1987), (Schubert, 1990).

$$f_{t-1} \wedge \neg f_t \supset \vee \{a_t \mid f \in \text{del}(a_t)\}$$

and

$$\neg f_{t-1} \wedge f_t \supset \vee \{a_t \mid f \in \text{add}(a_t)\}$$

6. The original formulation of SATPLAN also had clauses that said that only one action occurs at a time. For all actions a and b , the following holds,

$$\neg a_t \vee \neg b_t$$

However, later versions did away the exclusion between actions and borrowing from Graphplan, generated encodings from the planning graph that did allow parallel actions.

10.3.2 Encoding the Planning Graph

The planning graph can be expressed as a SAT problem in a straightforward fashion (Kautz et al., 1996). The fluents in the initial layer are clauses in the encoding, and the rest of the encoding process begins at the goal layer.

1. Fluents from the initial layer P_0 . If $f_0 \in P_0$ then f_0 is a clause, else $\neg f_0$ is a clause.
2. Fluents from the goal layer P_k . We add a fluent f_k for every goal proposition $f_k \in G$. Observe that the goal G is usually incompletely specified. We only add the known fluents to the SAT encoding.
3. An action implies its preconditions. If an action holds (is true) at layer/time t then its preconditions must hold at time $(t - 1)$ ¹². This is expressed as a clause $(a_t \supset \text{precond}(a)_{t-1})$ where $\text{precond}(a)$ is the conjunction of the preconditions of action a . For example we have, $(\text{UnSt}(A, B, t) \supset (\text{On}(A, B, t - 1) \wedge \text{ArmEmpty}(t-1) \wedge \text{Clear}(A, t - 1)))$.
4. For every fact f_t in level t , a disjunction of the actions that have f in their add effects is implied. For example, in some layer t it might be, $\text{Holding}(A, t) \supset \text{PkUp}(A, t) \vee (\text{Unstack}(A, B, t) \vee (\text{Unstack}(A, C, t) \vee \text{No-op})$. The actions that participate in the disjunction are determined by the planning graph.
5. Actions that are mutex define clauses in the SAT. For example in the planning graph of Figure 10.4, we have, $(\neg \text{PkUp}(C, 0) \vee \neg \text{UnSt}(A, B, 0))$.

Note that going backwards from the goal layer in the planning graph, an action only implies its preconditions and not its effects. Also a fact (fluent) implies an action that achieves it, in a manner similar to that of the explanatory *Frame* axioms. As reported in (Kautz and Selman, 1999), the above encoding of the planning graph is followed by a logical simplification algorithm that runs in polynomial time. This algorithm converts the encoding into CNF (Conjunctive Normal Form) and simplifies it by doing a limited amount of logical inference for mutex propagation. Kautz and Selman observe that the “*use of an intermediate plan graph representation appears to improve the quality of automatic SAT encodings of STRIPS problems.*”

10.3.3 Encoding Causal Links

In Chapter 7, we had looked at an approach for searching for plans in the plan space. The basic idea behind the partial order planning systems was to identify flaws in a partial plan and find operators to resolve those flaws. The flaws were of two kinds. The first, open goals, for which we needed either to find an existing action in the plan to support them, or insert a new action into the plan. The second, threats, which could possibly disrupt a support for a goal, had to be dealt with by “separating” the threatening action, or moving it before or after the threatened link.

The conditions required by a partial plan to be a valid one can be encoded in a SAT problem as well (Kautz et al., 1996). Kautz, McAllester and Selman report the encoding of a causal link planner SNLP (McAllester and Rosenblitt, 1991).

Let us say that we want to solve the Sussman’s anomaly in the blocks world,

Start state $S_0 = \{On(C, A), OnT(A), OnT(B), Clear(B), Clear(C), ArmE\}$

Goal $G = \{On(A, B), On(B, C)\}$

The set of possible ground actions is,

$A = \{UnSt(A, B), UnSt(A, C), UnSt(C, B), UnSt(C, A), UnSt(B, A), UnSt(B, C), Stack(A, B), Stack(A, C), Stack(C, B), Stack(C, A), Stack(B, A), Stack(B, C), PkUp(A), PkUp(B), PkUp(C), PtDn(A), PtDn(B), PtDn(C), No-op\}$

The set of ground fluents is,

$F = \{Clear(A), Clear(B), Clear(C), On(A, B), On(B, A), On(A, C), On(C, A), On(B, C), On(C, B), ArmE, OnT(A), OnT(B), OnT(C), Hold(A), Hold(B), Hold(C)\}$

We also have the two special actions A_0 , which produces the start state, and A_∞ which consumes the goal predicates. In plan space planning, we begin with the initial partial plan containing just these two actions, (A_0, A_∞) (see Chapter 7). The preconditions of A_∞ are the initial flaws, or open goals.

Let us say that we are looking for a set of actions $S = \{s_1, s_2, \dots, s_k\}$, called *step names*, with no *a priori* temporal ordering constraints on these actions, except that they occur after A_0 and before A_∞ . We know that for the Sussman’s anomaly problem $k = 6$. In practice, one could follow an iterative deepening approach to find a plan with the smallest number of actions.

A ground causal link is of the form $CausalLink(s_i, f, s_j)$ where $f \in precond(s_j)$ is a precondition of s_j and is produced by s_i , $f \in add(s_i)$.

A valid plan is an assignment of ground actions $A_i \in A$ to the step names $s_j \in S$ such that,

1. Every goal in the partial plan has a supporting action. That is, if f is a precondition (open goal) of s_j then there exists a causal link of the form $CausalLink(s_i, f, s_j)$.
2. Every causal link is *true*. If a plan contains $CausalLink(s_i, f, s_j)$ then,
 - (a) $f \in add(s_i)$.
 - (b) $s_i \prec s_j$, that is s_i happens before s_j .
 - (c) for every $s_k \neq s_i \neq s_j$ if $f \in del(s_k)$ then either $s_k, \prec s_i$ or $s_j, \prec s_k$. That is, no other action clobbers the causal link.
3. The ordering constraints are consistent. If $s_i, \prec s_j$ and $s_j, \prec s_k$ then $s_i, \prec s_k$ and it is never the case that $s_k, \prec s_i$.

For the Sussman's anomaly, we expect to find the following causal links to start with,

$$CausalLink(Stack(A, B), On(A, B), A_\infty)$$

and

$$CausalLink(Stack(B, C), On(B, C), A_\infty)$$

Each of the two actions $Stack(A, B)$ and $Stack(B, C)$ will have preconditions that will generate open goals that will need causal links of their own, which we expect to be found during the planning process. The following is an encoding of the planning problem as a SAT problem. The clauses of the SAT are defined as,

1. $(s = A_1 \vee s = A_2 \vee \dots \vee s = A_m)$ for all $s \in S$, and $A_i \in A$
2. $\neg(s = A \wedge s = B)$ for all $s \in S$, $A, B \in A$ and $A \neq B$
3. $\neg Adds(A_0, \Phi)$ $\Phi \in (F \setminus S_0)$
4. $Needs(A_\infty, \Phi)$ $\Phi \in G$
5. $Adds(s, \Phi) \equiv (s = A_1 \vee s = A_2 \vee \dots \vee s = A_n)$ where $s \in S$, $A_i \in A$, and $\Phi \in add(A_i)$
6. $Dels(s, \Phi) \equiv (s = A_1 \vee s = A_2 \vee \dots \vee s = A_o)$ where $s \in S$, $A_i \in A$, and $\Phi \in del(A_i)$
7. $Needs(s, \Phi) \equiv (s = A_1 \vee s = A_2 \vee \dots \vee s = A_p)$ where $s \in S$, $A_i \in A$, and $\Phi \in precondition(A_i)$
8. $Needs(s, \Phi) \supset (CausalLink(p_1, \Phi, s) \vee \dots \vee CausalLink(p_m, \Phi, s))$ where $p_i \in A \cup \{A_0\}$, $s \in A \cup \{A_\infty\}$ and $\Phi \in F$
9. $CausalLink(p, \Phi, s) \supset Adds(p, \Phi)$ where $p \in A \cup \{A_0\}$, $s \in A \cup \{A_\infty\}$ and $\Phi \in F$
10. $(CausalLink(p, \Phi, s) \wedge Dels(r, \Phi)) \supset (r \prec p \vee s \prec r)$ where $p \in A \cup \{A_0\}$, $s \in A \cup \{A_\infty\}$, $r \in A \setminus \{p, s\}$ and $\Phi \in F$
11. $A_0 \prec s$ $s \in A$
12. $s \prec A_\infty$ $s \in A$
13. $\neg(s \prec s)$ $s \in A$
14. $(p \prec r \wedge r \prec s) \supset (p \prec s)$ $p \in A \cup \{A_0\}$, $r \in A$, $s \in A \cup \{A_\infty\}$

The clauses in set 1 are a kind of closure axioms which say that the only actions are the one mentioned in the set A . The second set says that each step is unique and distinct. The third clause says that the initial action only produces the propositions in the start state and nothing else, and the fourth axiom states that in a successful plan, the goal propositions are "needed" by the final action. The sets of clauses 5, 6 and

7 define the relations *Adds*, *Dels* and *Needs*, between propositions and corresponding actions. The clauses in set 8 say that whenever a proposition is “needed” by an action, there should be a causal link in which that proposition is produced. The set 9 says that every causal link *CausalLinki* (p, Φ, s) implies the relation *Adds*(p, Φ). Ten says that if action r is a threat to the above causal link, then it must either happen before p or after s . Clauses 11 and 12 place the initial and the final action at the two ends, clause 13 says that an action cannot be in a ordering relation with itself, and clause 14 says that the ordering relation between actions is transitive.

The SAT encoding above uses ground instances of fluents and actions. It is possible to arrive at a *lifted* encoding in which variables are used for fluents and actions, which has a smaller number of clauses. The interested reader is referred to (Kautz et al., 1996).

10.4 Heuristic Search

In Chapter 7, we briefly looked at the idea of doing state space search for planning. In Chapter 3, we have looked at the idea of using heuristic functions to guide search. The heuristic functions we described there were static, in the sense that they derived a distance estimate by looking at the current state and the goal state. These functions were domain dependent and the user was expected to define them.

More recently, the idea of doing some analysis in a domain independent fashion has emerged. The idea is that for problems which are intrinsically hard, one can relax some constraints and solve a simpler version of the problem to estimate the cost of solving the original problem. Then, in a state space search scenario, one can do this analysis for all the candidate nodes and in effect compute a heuristic value for each state in a domain independent fashion. As shown in Figure 10.15 below, this heuristic function can help guide the search process in selecting the nodes most likely to lead to the goal. Thus, heuristic search planning is also a two stage process in which one alternates between a node *selection* phase (by solving the simpler problem) and a node *expansion* phase which generates the candidates. The key thing is that the secondary search on the simpler problem should be significantly inexpensive, preferably being a low order polynomial in complexity.

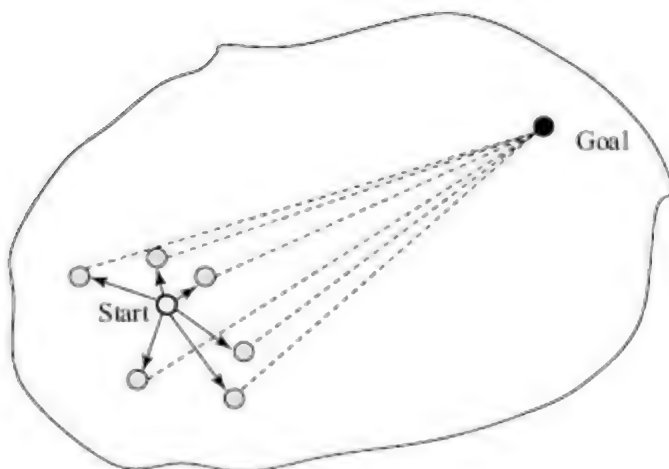


FIGURE 10.15 (Reproduced from Figure 2.2). The heuristic function estimates the distance to the goal.

The simpler problem to solve is usually a transformation of the planning problem into another planning problem in which the operators do not have delete effects.

Given the planning problem P defined as a triple (S, G, O) where,

- S is set of facts completely describing the Given or Start state,
- G is a set of facts required to be true in a goal state, and
- O is the set of operators

we can define a new planning problem P' by replacing the set of operators O with a set O' . The set O' contains the corresponding operators as the set O , except that there are no delete effects in any operator. For example, the operator *PickUp*,

```
(:action pickup
  :parameters    (?x -block)
  :precondition  (and (onTable ?x) (armempty) (clear ?x))
  :effect (and    (not (armempty)) (holding ?x) (not (onTable ?x))))
```

is replaced by the operator *PickUp-R*,

```
(:action pickup-r
  :parameters    (?x -block)
  :precondition  (and (onTable ?x) (armempty) (clear ?x))
  :effect        (holding ?x))
```

where the negative or delete effects have been removed. The new problem P' is known as the *relaxed planning problem*. The solution to the relaxed planning problem is the *relaxed plan*.

Observe that certain combinations of actions in the relaxed plan may not be allowed in the original planning problem. For example, we can

have *PickUp-R(A)* followed by *PickUp-R(B)*, or even have both the actions in parallel, in the relaxed plan. This happens because the precondition “*armempty*” is no longer deleted in the relaxed domain. Nevertheless, one expects that a solution to the relaxed problem will give us some idea about the cost of the solution in the original domain. Enough, hopefully for the search algorithm to be able to choose the next action in a more informed manner. In general, the optimal relaxed plan would be shorter than the optimal plan in the original domain. This is because additional steps may be needed to compensate for ignoring the delete actions. For example, the sequence (*PickUp(B)*, *Stack(B, C)*, *PickUp(A)*) might be part of a plan in the domain in which only (*PickUp-R(B)*, *PickUp-R(A)*) might be part of a relaxed plan. In the rest of the discussion, we omit the suffix *r* for the relaxed actions.

If we could find an optimal plan for the relaxed problem then we could use the length¹³ of the solution as an admissible heuristic for an algorithm like A*. Unfortunately, the problem of finding the optimal solution to the relaxed plan is NP-hard (Bylander, 1994). Therefore, one has to instead work with an estimate of this value. Two well known heuristic search planners *HSP* and *FF*, computed this estimate in different ways.

10.4.1 HSP

The Heuristic Search Planner by Blai Bonet and Hector Geffner (2000; 2001; 2001a) estimates the distance to the goal from a given state by estimating the number of steps, or the sum of their costs, for *each proposition* in the goal description.

Given a planning problem, the state space is defined by the set of possible states. Each state is a set of ground propositions. Each instance *a* of the STRIPS actions has a set of preconditions *precond(a)*, add effects *add(a)*, and delete effects *del(a)*. In the relaxed version of the problem, the delete effects are ignored. The estimated cost *g(s, p)* of reaching a proposition *p* from a given state *s* is defined as,

$$\begin{aligned}
 g(s, p) &= 0 && \text{if } p \in s \\
 &= \min_{a \in O(p)} [1 + g(s, \text{precond}(a))] && \text{otherwise}
 \end{aligned}$$

```

CostP (state = S, propositions = P, actions = A)
1  for each p ∈ P
2      if p ∈ S then g(s, p) ← 0
3      else g(s, p) ← ∞
4  availableActions ← true
5  while availableActions = true
6      a ← Applicable(A, S)
7      if a = "nix" then availableActions ← false
8      else A ← A \ a
9          for each p ∈ add(a)
10             S ← S ∪ {p}
11             g(s, p) ← min(g(s, p), CostS(precond(a), g(s, P)))
12
13  return g(s, P)
CostS (preconditions = C, g(s, P))
1  cost ← 0
2  for each p ∈ C
3      cost ← cost + g(s, p)
4  return cost

```

FIGURE 10.16 The procedure *CostP* initializes cost of each predicate $g(s, p)$ to 0 or infinity. It then applies actions that are applicable. Each applied action augments costs of $g(s, p)$ for every add effect p of the action, and adds the effect to the state, possibly triggering more actions. The aggregation function *CostS* shown here adds the cost of individual elements of the set. The function *CostP* returns an array $g(s, P)$ with a cost associated with each proposition $p \in P$. We assume a function *Applicable*(A, S) that returns an action $a \in A$, if it is applicable in S , and returns *nix* if there is no such action.

where $O(p)$ stands for the set of actions that have p in their add effects. Note that the function g is defined, both for a proposition as well as for a set of propositions. The function for a set is some kind of an aggregation over the individual components of that set.

The algorithm used in HSP updates the values $g(s, p)$ using a forward chaining procedure *CostP* described in Figure 10.16. The algorithm initializes the cost of each proposition to 0 if it belongs to the state s , else it sets it to a very large value, ∞ . It then picks an action that is applicable, adds its effects to the state s , and re-computes cost for each added effect. When it terminates, *CostP* returns a cost value for each proposition. This cost is the estimated cost of reaching the proposition from the given state s . If the cost of a proposition remains ∞ , it means that the proposition is not reachable in the domain. The aggregation function *CostS* here computes the cost of a set of propositions (the preconditions of the action) as the sum of the costs of each member. Note that these costs are all finite, since they pertain to the preconditions which have been added to the state s at some time before the action becomes applicable. Other methods of aggregation, for example using the maximum value, could be used.

Given a state s , the estimated cost $h(s)$ of achieving the goal G from s is defined as,

$$h(s) \stackrel{\text{def}}{=} g(s, G)$$

where $g(s, G)$ is an aggregation of all $g(s, p)$ such that $p \in G$. This

aggregation function may be defined in different ways, but the two most commonly used aggregations are the *additive* and the *max* heuristic. We can call the corresponding heuristic functions as $h^+(s)$ and $h^{max}(s)$.

The additive heuristic $g^+(s, G)$ computes the value as a sum of the costs of each proposition in the goal.

$$g^+(s, G) = \sum_{p \in G} g(s, p)$$

The additive function is the function *CostS* shown in Figure 10.16. The additive function assumes that each goal proposition p is arrived at by an *independent* set of actions. This assumption of independence is however not always true, because some actions may generate more than one proposition. As a result, the estimated cost using the additive function may be more than the actual cost, and therefore the additive function is not guaranteed to be admissible (see Chapter 5 for a discussion on admissibility).

The other commonly used function is $g^{max}(s, G)$ which aggregates the value for the set by taking the maximum of the constituent costs.

$$g^{max}(s, G) = \text{Max}_{p \in G} g(s, p)$$

The max heuristic can be thought of as making a complete dependence assumption, which assumes that the sequence of actions that achieves the costliest proposition also achieves the other propositions on the way. Obviously, even this assumption is not likely to hold, but the max heuristic has the property of being admissible because it never overestimates the cost. Consequently, if one has the task of finding the optimal (or shortest where costs are uniform) plan then one should use the max heuristic for aggregation. On the other hand, the additive heuristic is likely to be more informed because it looks at all the constituent propositions, leading to a more focused search.

The original version of *HSP* used the additive heuristic with Hill Climbing (see Chapter 3). A later version *HSP2* uses the weighted A^* algorithm (see Chapter 5), and can be made an admissible algorithm with an appropriate choice of heuristic function and the weights. *HSP2* also used a variation on the heuristic function called $h^2(s)$. Instead of taking only the costliest proposition, as is done in $h^{max}(s)$, the $h^2(s)$ function looks at the costliest *pair of propositions* that are achieved at the same time. Let us say that the two propositions are called p and q . Then the heuristic function $h^2(s)$ is defined as (Ghallab et al., 2004),

$$\begin{aligned}
g^2(s, p) &= 0 && \text{if } p \in s \\
&= \min_{a \in O(p)} [1 + g^2(s, \text{precond}(a))] && \text{otherwise} \\
g^2(s, \{p, q\}) &= 0 && \text{if } p, q \in s \\
&= \min \{ \min_a [1 + g^2(s, \text{precond}(a)) \mid \{p, q\} \subseteq \text{add}(a)], \\
&\quad \min_a [1 + g^2(s, \{q\} \cup \text{precond}(a)) \mid p \in \text{add}(a)], \\
&\quad \min_a [1 + g^2(s, \{p\} \cup \text{precond}(a)) \mid q \in \text{add}(a)] \} \\
g^2(s, G) &= \max_{p, q} \{ g^2(s, \{p, q\}) \mid \{p, q\} \subseteq G \}
\end{aligned}$$

and,

$$h^2(s) = g^2(s, G)$$

Clearly, $h^2(s)$ is more informed than $h^{\max}(s)$ which can be thought of as $h^1(s)$. In fact, we can define a family of heuristic functions $h^1(s)$, $h^2(s)$, ..., $h^m(s)$ which are increasingly more informed, as well as increasingly more expensive to compute. One can observe that when for $|G| = j$, the function $h^j(s)$ in fact solves the relaxed planning problem, which is NP-hard.

Another version *HSPr*, a regression planner, searches backward from the goal propositions (see Chapter 7 for a discussion on backward state space search). According to Bonet and Geffner, heuristic computation in *HSP* and *HSP2* takes up about 80% of the total computation time, and this results in fewer nodes being explored by the search algorithm. The problem becomes acute because as the forward state-space search generates new candidate nodes as it progresses, the heuristic value for every new state has to be computed.

The algorithm *HSPr* instead searches in the *regression space*, searching backwards from the goal propositions. A major advantage of doing so is that the heuristic computation is drastically reduced. Figure 10.17 illustrates the situation. The *HSPr* algorithm initially computes the value $g(s_0, p)$ for every ground predicate p , where s_0 is the start state. This is illustrated in the figure by the dashed lines connecting the nodes, which are drawn when they first appear as time increases from left to right (as in the planning graph). Since we are ignoring the delete effects, once a proposition has been generated it is always a part of all subsequent states. The length of each dashed line then is a measure of $g(s_0, p)$. Now when the backward search regresses over the goal set, it produces candidate subgoal sets. In the figure, we illustrate two candidate sets *SG1* and *SG2* shown in shaded areas. The heuristic values for these subgoal sets can simply be aggregated from the values of the individual propositions which have been precomputed. Observe that the $g(s_0, p)$ is always measured with respect to the start state s_0 which is the destination of backward search. For example, if we were using the h^{\max} then $h^{\max}(\text{SG1}) = 3 + 2 + 3 = 8$ and $h^{\max}(\text{SG2}) = 3 + 2 + 1 = 6$.

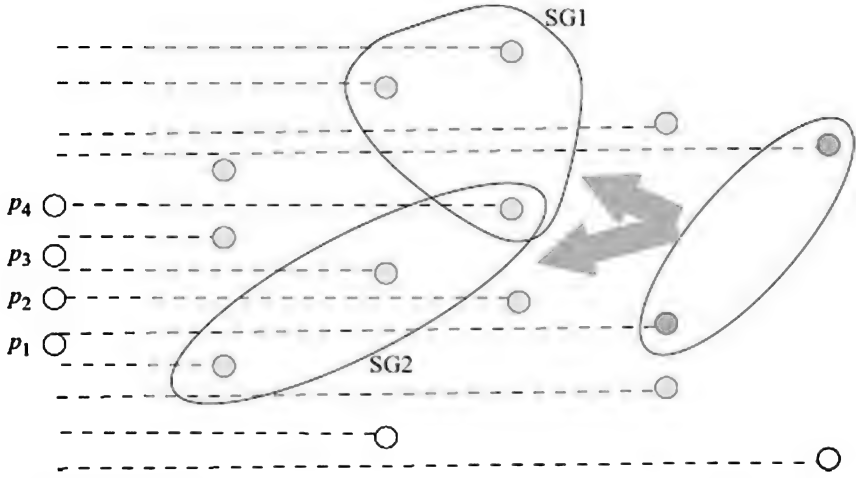


FIGURE 10.17 *HSPr* precomputes $g(S_0, p)$ for every p , represented by dashed lines above. Then during a backward search, it needs to only aggregate the values for each proposition in a subgoal set. The dark nodes represent the goal and the two sets of three nodes represent the two subgoal sets.

Another advantage of using backward search is that the branching factor is likely to be lower as compared to forward search as discussed in Chapter 7. However, as also described there, backward search has a disadvantage that it can generate spurious states, for example a blocks world subgoal in which the robot arm is holding two different blocks, or two different blocks are on a third block. Clearly, this is not possible and exploring such goals is an exercise in futility.

HSPr strives to catch *some* of these spurious subgoals by computing a binary mutex between some pairs of propositions, in a manner similar to as is done while constructing the planning graph in *Graphplan*. The definition of mutex used in *HSPr* is (Bonet and Geffner, 2001).

Given an initial state s_0 and a set of ground operators A , a set M of pairs of propositions is a mutex set *iff* for all pairs $R = \{p, q\} \in M$.

1. Both p and q are not true in s_0 , and
2. For every action $a \in A$ such that $p \in \text{add}(a)$ (and vice versa),
 - (a) either $q \in \text{del}(a)$ or
 - (b) $q \notin \text{add}(a)$ and for some $\text{precond}(a)$, $R' = \{r, q\}$ is a pair in M

The procedure for constructing the set M began by constructing a set $M_0 = M_A \cup M_B$ where,

- M_A is the set of pairs $P = \{p, q\}$ where for some action a , $p \in \text{add}(a)$ and $q \in \text{del}(a)$,
- M_B is the set of pairs $P = \{r, q\}$ such that for some pair $P' = \{p, q\} \in M_A$ and some action a , $r \in \text{precond}(a)$ and $p \in \text{add}(a)$.

The algorithm for computing M begins with the set M_0 as defined above and removing all “bad pairs” that do not satisfy either of condition 1 and 2 defined above. Like in *Graphplan*, not all sets of mutually exclusive pairs are identified here, but the ones that are identified do help in pruning the search space. *HSPr* uses the additive heuristic h^+ and the set M to guide the backward search, which is the weighted A^* search¹⁴ with the weight w set to 5, like in *HSP2*. Bonet and Geffner report that in some domains, *HSP2* had better performance, while in some *HSPr* was better. An effort to exploit the benefits of backward search with an *FF* styled heuristic is described in (Kumashi and Khemani, 2002).

10.4.2 FF

The algorithm Fast Forward (FF), also a forward state space search, employed a slightly different algorithm and a different strategy for computing the heuristic function (Hoffmann and Nebel, 2001), (Hoffmann, 2001).

To compute a heuristic value for a given state s , FF constructs a *relaxed planning graph* from that state s . The relaxed planning graph is the planning graph constructed by ignoring the delete effects in actions. Since the delete effects are ignored, the resulting planning graph has no mutexes at all, because the mutexes are a direct consequence of having delete effects. Consequently, when the goal propositions are produced, the plan extraction phase can find a plan without the need to backtrack. This is because *any* set of actions that achieves a goal/subgoal will be nonmutex. This means that a solution to the relaxed planning problem can be found in polynomial time (in terms of the number of propositions in the initial state, the number of planning operators, and the maximum size of the add list for these operators).

Let the solution to the relaxed planning problem starting at state s be (A_1, A_2, \dots, A_m) , where each A_i is the set of actions in the i^{th} layer that are part of the solution, and the goal propositions first appear in the m^{th} layer. Then the heuristic used by FF h^{FF} simply counts the number of actions in the relaxed plan found.

$$h^{FF}(s) = \sum_{i=1, \dots, m} |A_i|$$

Like *HSP*, the algorithm *FF* too does forward state space search and selects the node to expand, based on the heuristic values of the candidates. The best candidate node should ideally have the lowest value. However, the fact that the plan extraction phase happens without running into mutexes implies that it is not guaranteed that the plan extracted is the optimal one. In fact, as observed above, the task of finding the optimal plan is NP-hard. *FF* uses some heuristics while searching backwards to try and find the best plan. These heuristics are,

1. *Prefer No-op Actions*. If a *No-op* operation is available for supporting a proposition p then it is selected. In fact, this heuristic

was also used in *Graphplan*, which first tries *No-ops*.

2. *Difficulty Heuristic*. If a *No-op* action is not available to support the proposition, one of the other actions that produce p must be chosen. The difficulty heuristic says choose an action whose preconditions are the “easiest” to solve. The “difficulty” of an action is defined as follows,

$$\text{difficulty}(a) = \sum_{q \in \text{precond}(a)} \min\{i \mid \text{proposition } p \text{ occurs in layer } i\}$$

Given a choice of actions that achieve p , FF picks the one with the lowest difficulty value. The difficulty value for each action can be computed when it is first inserted in the planning graph. The reader will observe the similarity with the $h^+(s_0, g)$, used by *HSP_r* with g corresponding to $\text{precond}(a)$.

Hoffmann and Nebel (2001) report that they implemented their own version of *Graphplan* for solving the relaxed planning problem, without having to worry about mutex relations. When the planning graph is being constructed, each action and each proposition is marked with the layer in which it first appears. Armed with this information, the plan extraction phase can apply the above heuristics efficiently.

The forward search used by FF is a greedy algorithm that is a variation of Hill Climbing, called Enforced Hill Climbing (EHC). The EHC algorithm is designed to work in domains in which from any given state, a better node is at most a few steps away. The basic idea is to search in a breadth first fashion till a better node is found, and when one is found, the algorithm moves to that node in a greedy manner. The algorithm, given in Figure 10.18, maintains a hash table of CLOSED nodes, and does not visit the same nodes again.

```

EnforcedHillClimbing (state = s, goal = G, actions = A)
1  plan ← ( )
2  closed ← ( )
3  current ← s
4  h(current) ← HeuristicFF(current, Goal)
5  while h(current) ≠ 0
6      closed ← closed ∪ {current}
7      planSegment ← NextBetter(current, closed, A)
8      if planSegment = "nix"
9          then return failure
10     else plan ← append (plan, planSegment)
11         current ← Last(current, planSegment)
12         h(current) ← heuristicFF(current, Goal)
13         closed ← HashInsert(closed, planSegment)
14  return plan

```

FIGURE 10.18 The procedure *EnforcedHillClimbing* synthesizes a plan using a greedy approach moving to the first better node it finds. We assume the following functions: *HeuristicFF*(current, Goal) computes the heuristic function as described in the text; *NextBetter*(current, closed, A) finds the closest node that is better than the current node using breadth first search, avoiding the nodes in *closed*; *Last*(planSegment) progresses over the plan

segment found and returns the final state reached; and *HashInsert(closed, planSegment)* updates the table of closed nodes seen by *EHC*.

The algorithm *EHC* commits to a better state as soon as it finds it. This may mean in some domains that it lands up in a dead end from where no moves are possible. Hoffman and Nebel observe that if *EHC* is going to fail, it fails pretty quickly. This means that FF can try something else to find a plan. After trying out some approaches based on randomization, they settled for a second stage in which they use a complete algorithm like Best First Search (see Chapter 3).

One of the drawbacks of forward state space search is that the number of actions applicable in a given state may be quite large, as illustrated in Figure 7.4. Many of these actions may not be relevant to the goal being solved. While we do expect the heuristic function to select the better options from the set of candidates, computing the heuristic value for each such candidate adds to the computational overheads. FF has the option of using some pruning techniques which are quite effective, but not always be sound. This means that they may prune away a candidate node that would have been part of a solution.

One such pruning method is the use of *helpful actions*. The set of helpful actions is the set A_1 in the relaxed plan (A_1, A_2, \dots, A_m) found by *FF*. The pruning method used is to consider *only* the actions in the helpful actions set.

The method can be very effective in many situations. Consider for example the planning problem in which there are a hundred blocks $\{A_1, A_2, \dots, A_{100}\}$ stacked on another hundred corresponding blocks $\{B_1, B_2, \dots, B_{100}\}$, and the goal is to achieve $On(A_{21}, A_6)$. Then the relaxed plan found would have only the action $Unstack(A_{21}, B_{21})$ in the first layer of actions, and *FF* would only consider that and ignore the ninety nine other unstack actions.

However, consider the goal $On(B_{21}, B_6)$. *FF* would begin by finding the relaxed plan,

$$(\{Unstack(A_{21}, B_{21}), Unstack(A_6, B_6)\}, \{Pickup(B_{21})\}, \{Stack(B_{21}, B_6)\})$$

This plan has two helpful actions, and let us say that *FF* chooses $Unstack(A_6, B_6)$. This results in a new state in which $Holding(A_6)$ and $Clear(B_6)$ are true, and $ArmEmpty$ is false. From this state, there are several relaxed plans possible. The important goal in all these relaxed plans is to achieve $ArmEmpty$ so that $Pickup(B_{21})$ becomes possible. We look at 3 of the 101 such actions possible, yielding the following relaxed plans.

$$\begin{aligned} \text{relaxed-plan-1} &= (\{PutDown(A_6)\}, \{Unstack(A_{21}, B_{21})\}, \\ &\quad \{Pickup(B_{21})\}, \{Stack(B_{21}, B_6)\}) \\ \text{relaxed-plan-2} &= (\{Stack(A_6, A_{21})\}, \{Unstack(A_{21}, B_{21})\}, \\ &\quad \{Pickup(B_{21})\}, \{Stack(B_{21}, B_6)\}) \\ \text{relaxed-plan-3} &= (\{Stack(A_6, B_6)\}, \{Unstack(A_{21}, B_{21})\}, \end{aligned}$$

$$\{Pickup(B_{21}), \{Stack(B_{21}, B_6)\}\}$$

As far as the relaxed problem is concerned, all three actions $PutDown(A_6)$, $Stack(A_6, A_{21})$ and $Stack(A_6, B_6)$ achieve *ArmEmpty*. The fact that the latter two have undesirable effects is not noticed because those effects are the delete effects which are ignored. Thus, FF cannot “see” that $Stack(A_6, A_{21})$ will delete $Clear(A_{21})$ needed for $Unstack(A_{21}, B_{21})$, and $Stack(A_6, B_6)$ will delete $Clear(B_6)$ needed for $Stack(B_{21}, B_6)$. As we can see, if FF uses the *relaxed-plan-2* or *relaxed-plan-3*, the set of helpful actions would not have been helpful at all. One method suggested in (Hoffmann and Nebel, 2001) is to consider the union of actions in all relaxed plans as helpful actions.

Another heuristic called *added goal deletion* works as follows. Let us say that a candidate state s has achieved a goal proposition $p \in G$. Now, FF constructs a relaxed plan P' starting from the state s in order to evaluate the heuristic value, $h^{FF}(s)$. If this plan P' has a relaxed action a' such that the corresponding action a has the goal proposition p in its delete effects then the added goal deletion heuristic says that state s should not be selected by enforced hill climbing. As an example, consider the goal $\{On(A, B), On(B, C)\}$ when all three blocks A , B and C are on the table in the start state. The relaxed plan contains picking up blocks A and B , and stacking them on B and C respectively.

$$relaxed-plan = (\{PickUp(A), PickUp(B)\}, \{Stack(A, B), Stack(B, C)\})$$

Let us say the FF picks the (helpful) action $PickUp(A)$, and in the next step follows it up with $Stack(A, B)$ resulting in state s . Note that after picking up A , it needs to achieve *ArmEmpty* before it can pickup B , and that is why $PickUp(B)$ is not the second action, and $Stack(A, B)$ is. The resulting state $s = \{On(A, B), Clear(A), OnT(C), Clear(C), Armempty\}$ achieves the goal proposition $On(A, B)$. But the relaxed plan from state s is,

$$relaxed-plan = (\{Unstack(A, B)\}, \{Stack(A, B), Pickup(B)\}, \{Stack(B, C)\})$$

which contains the action $Unstack(A, B)$ which deletes the goal that was achieved in s , that is $On(A, B)$. Thus, using the added goal deletion heuristic state s is not selected by EHC. Observe that having picked up A , FF needs to do something with it. If it were using the helpful actions heuristics, it would have pruned away the $PutDown(A)$ action, and would then get stuck holding block A . One way of avoiding this kind of a deadlock is to try and do some pre-processing to determine a goal ordering in advance, and then have the EHC algorithm operate on the goals incrementally in the given order. This would need some heuristics to decide upon the goal ordering.

To cater to some of the cases where FF misses out on the desired

actions and commits to a wrong set of actions, it has a built-in restart facility. In the breadth first mode, if the number of successors who are not better exceeds a threshold, the search aborts and restarts. Given the memory of closed nodes that it keeps, the new explorations will be in a different direction.

10.4.3 Fast Downward

The algorithm *Fast Downward* by Malte Helmert combines the forward search of *FF* with a new heuristic based on *causal graph* analysis, and is described briefly below. For more details, the interested reader is referred to (Helmert, 2004; 2006; 2008).

The *Fast Downward* program also extends the richness of domain a planner can handle. Like metric *FF* (Hoffmann, 2003), it could also handle metric resources and conditional effects. Metric resources are fluents that can take numerical value, for example temperature, capacity in a parking lot, or amount of fuel in the tank. Handling metric resources involves handling arithmetic expressions, comparing quantities (for example fuel > 10), and assigning new values in the effects of the action (for example fuel \leftarrow fuel - distance*rate). Extending to metric resources is natural for *Fast Downward* because the representation it uses is the state-variable representation, referred to as *multi-valued planning tasks (MPT)* in the associated literature. The program also handles derived predicates and conditional effects, both in a similar manner. Derived predicates are predicates that are not modified by actions, but depend upon the values of some other predicates. This relation is captured in *Fast Downward* by axioms of the form $\langle \text{cond}, v, d \rangle$, where *cond* is a pattern or condition defined on some variables, and when the condition holds in a state, the derived predicate *v* gets a value *d*. The same format used in the effects of an action describes conditional effects.

Helmert believes that the STRIPS like representation obscures important causal structure in the domain, and that ignoring the delete lists for estimating the heuristics, ignores some vital information. *Fast Downward* first converts problems in PDDL into the MPT representation described below. Then, it compiles state transition and causal dependencies into separate graphs, and computes the heuristic values over these graphs. Finally, it uses the heuristic value to guide forward search.

A planning problem in the MPT representation is a 5-tuple $\langle V, s_0, s^*, D, A \rangle$ where¹⁵,

- *V* is a set of (multi-valued) variables *v* each with its domain D_v . The variables are partitioned into two sets. One containing fluents, including numeric fluents, which are changed in the effects of operators in *A*. The second, called derived variables, are determined by other variables, and are computed using derived axioms in *D*.

The domains of derived variables contain an *undefined value* \perp .

A *partial variable assignment* or a *partial state* over V is a function s on some subset of V , such that $s(v) \in D_v$ wherever $s(v)$ is defined.

- s_0 is a state over V called an initial state.
- s^* is a partial state over V called the goal.
- D is a set of axioms over V . Axioms are of the form $\langle \text{cond}, v, d \rangle$ where cond is a partial variable assignment called the *condition* or *body* of the axiom, v is a derived variable, and $d \in D_v$ is a value of the variable that is assigned when the condition is true. The pair (v, d) is called the *head* of the axiom.
- A is a set of actions of the form $\langle \text{precond}, \text{effects} \rangle$, where precond is a partial variable assignment, and effects contains a set of effects of the form $\text{effect} = \langle \text{cond}, v, d \rangle$. The effect assigns the value $d \in D_v$ to the variable v . If the condition in $\langle \text{cond}, v, d \rangle$ is non-empty then the effect is a conditional effect, affected only when the condition is true.

Fast Downward begins by creating some graphs described below. The aim is to be able to decompose the planning task into subtasks which can be evaluated independently to estimate the heuristic values. The *causal graph heuristic* estimates the cost of reaching the goal by solving a number of subtasks of the planning tasks by looking at segments of the *causal graph*. The costs are computed over *domain transition graphs*.

A *Domain Transition Graph (DTG)* for a variable $v \in V$, introduced by Jonsson and Bäckström (1998), is a representation of the ways in which the variable can change values. The nodes in the graph are the values in the domain of v , D_v . An edge from a value d to a value d' is added to $\text{DTG}(v)$ under following conditions.

- If v is a fluent then for each $\text{effect} = \langle \text{cond}, v, d' \rangle$ of an action a with precondition precond ,
 - if $\text{precond} \cup \text{cond}$ contains some condition $v = d$, an edge from d to d' labelled with $(\text{precond} \cup \text{cond}) \setminus \{v = d\}$ is added.
 - if $\text{precond} \cup \text{cond}$ does not contain the condition $v = d$ for any $d \in D_v$, an edge from each $d \in D_v \setminus \{d'\}$ labelled with $(\text{precond} \cup \text{cond})$ is added.
- If v is a derived variable then for axiom condition $\langle \text{cond}, v, d' \rangle \in D$, such that,
 - cond contains some condition $v = d$, an edge from d to d' labelled with $\text{cond} \setminus \{v = d\}$ is added.
 - cond does not contain the condition $v = d$ for any $d \in D_v$, an edge from each $d \in D_v \setminus \{d'\}$ labelled with cond is added.

Edges represent transitions and the labels represent conditions on

transitions. Observe that the labels are on other variables in the domain that must satisfy some conditions. The authors assume that the edges for transitions derived from operators have weight 1 and edges from axioms have weight 0. These weights play a role in the heuristic computation. Figure 10.19 illustrates the transition graph for a three block STRIPS problem. The graph is shown for the state variable $On(A)$ which can take values from its domain $\{B, C, nil\}$ where $On(A) = nil$ stands for the situation when the robot arm is holding block A, and $Holding(arm) = nil$ stands for *ArmEmpty*.

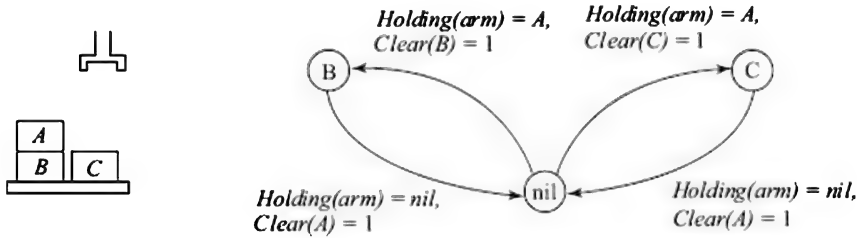


FIGURE 10.19 The domain transition graph for the state variable $On(A)$, between its three values B , C , and nil .

If all the actions in the MPT are *unary*, that is they have only one effect, then the transitions in the state space have a correspondence with transitions in the *DTG*. A given state in the state space can be represented by a set of *active* vertices in the *DTGs*. Applying an action corresponds to making a transition in one *DTG*. For non-unary actions, one has to link variables in the effects so that transitions in the transitions in their *DTGs* happen in a synchronized fashion. The treatment for handling transitions in derived variables involves extending the *DTGs* to states not described explicitly by the axioms, and take into account cascading effects (see (Helmert, 2006) for a detailed discussion).

Causal Graphs (CG) are designed to capture dependencies between variables (Williams and Nayak, 1997). Causal graphs capture relations between variables and are defined as follows. Let P be a planning problem with the set of state variables V . The causal graph $CG(P)$ for the planning problem is a directed graph with the variables as vertices. An edge exists between two vertices (v, v') iff $v \neq v'$ and one of the following conditions holds,

Transition Condition The domain transition graph of v' has a transition labelled with some condition on v .

Co-occurring effects The effects for some action a includes both v and v' .

An edge signifies the fact that changes in the target variable v' are dependent on changes in the source variable v . The first condition allows us to trace the set of variables, $ancestor(g)$ that could effect a goal

variable g . This is called the achievability definition of causal graphs. However, if a planner focuses only on $ancestor(g)$ it might be possible that some action in the plan may have an undesirable side effect on the goals. To cater to this possibility, the second condition introduces the *separability* definition in the causal graphs.

If the causal graph is acyclic and the domain transition graph is strongly connected, then the following algorithm can be used to find a plan. First, we identify a variable v in the causal graph with no outgoing variables, a *sink*. We check whether a goal is defined for this variable. If no, we remove the sink v from the graph, and work on the smaller problem. If yes, we find a path from $s_0(v)$ to $s^*(v)$ in the domain-transition graph. This gives us a “high level plan” in an abstraction space which can be refined further by setting the variables of the predecessors of v in the causal graph to the values required for the transitions in the high level plan, and working on those after removing v . This process of planning is basically *refinement* planning in which an abstract plan is refined into a more concrete one.

In practice, however, the causal graph for a given planning problem is not likely to be acyclic. The *Fast Downward* algorithm works on segments of the complete causal graph to derive the causal graph heuristics to guide a state space search. It first eliminates variables which do not belong to the set of ancestor variables (in the causal graph) for any goal variable. Then, it prunes the resulting causal graph to make it acyclic. This is done as follows.

First, the strongly connected components of the causal graph are identified. Each component is handled separately to remove cycles. The variables of each component are arranged in a total order $v \prec v'$ defined as follows.

1. Each variable v is assigned a weight n that counts the number of actions or axioms that induce v . The weight is an indicator of the importance of the variable.
2. A variable v with the lowest weight is fixed in the lowest position in the ordering. That is, for all other variables v' , $v \prec v'$.
3. Variable v is removed along with all its incident edges.
4. The remaining variables are processed in the same fashion.

When $v \prec v'$ holds, we say that v' has a *higher level* than v . In the pruned causal graph, we retain only those edges (v, v') where $v \prec v'$ holds. Next, in the domain transition graph, for each variable $DTG(v)$ all conditions on v' , where $v \prec v'$, are removed. After this, all the DTGs are simplified by removing dominated transitions. If t and t' are transitions on the same pair of values and the condition on t' is a subset of or equal to the condition on t then the transition t is removed from the DTG (since it may be more expensive to compute).

The basic idea behind computing the causal graph heuristic is

understood by viewing it as a bottom up process, with the lowest level variables being considered first. Conceptually, the task is to compute the cost for each transition in the DTG. These costs of the form $cost_v(d, d')$ are then used to estimate the cost of solving the goal. In practice, not all such costs need to be computed and the algorithm in *Fast Downward* takes a top down view of this process¹⁶. The bottom view is as follows. We first compute the costs for the lowest variables in the ordering, and use these costs to compute the costs for higher variables.

1. If a variable v has no causal predecessors (is at the bottom of the causal ordering), then $cost_v(d, d')$ is the length of the shortest path from d to d' in $DTG(V)$. In fact, the costs from each d to all, d' can be computed by Dijkstra's single-source shortest path algorithm (see (Cormen et al., 2001)).
2. Let V_v be the set consisting of v and all its predecessors in the pruned causal graph $CG'(p)$. Let Π_v be the planning task induced by V_v in which the initial value of v is set to d and the goal value set to d' .
3. $cost_v(d, d') = |\pi_v|$ where π_v is the plan found for Π_v (by the algorithm for acyclic causal graphs outlined above). This cost includes the cost of the low level actions needed by the high level plan.

The total cost of achieving the goal state from a given state s is defined as the sum of $cost_v(s(v), s^*(v))$ over all variables v for which the goal s^* is defined. In practice, the algorithm implemented computes costs in a top down fashion, making recursive calls to compute the edge costs that it needs. As described in (Helmert, 2006), the algorithm to compute the $cost_v(d, d')$ is basically Dijkstra's algorithm with embedded recursive calls, made in the top down fashion. The cost of traversing each arc in the $DTG(v)$ is the base cost (1 for fluents and 0 for derived variables) plus the cost of achieving the conditions associated with the transition. For a given condition $v' = e'$, the cost depends upon the value e in the state when the transition occurs. To facilitate that the algorithm commits greedily to whatever transition it can make to some value d' and annotates the resulting node d' with the local state achieved by the plan to reach d' .

Fast Downward has three different search algorithms implemented.

1. Best First Search Using the Causal Graph Heuristic

It also uses a notion of *preferred operators* similar to *FF*'s use of helpful operators, but *helpful transitions* which are generated by looking at the paths found in the DTGs. The algorithm maintains two OPEN lists, one for all operators and one for the preferred operators, and picks nodes alternately from the two lists. The algorithm also uses *deferred heuristic*

evaluation. Instead of evaluating all successors of a given node s , they are placed on the OPEN list with the heuristic value of s . The successors are evaluated only when they are picked from the OPEN list. This has the effect that if better successors are found early, their siblings are never actually evaluated. In fact, the successor itself is not put in the OPEN, but a pointer to the parent and the operator generating the successor is stored, and the successor is generated only when it needs to be evaluated.

2. Multi-heuristic Best First Search

This is a variation of best first search that allows the use of multiple heuristic functions. The different heuristic values are not combined into one value. Instead, separate OPEN lists are maintained for each heuristic function. The search algorithm picks nodes for expansion alternately from the lists. When successors are generated, they are evaluated by the different functions and inserted into the respective lists. The idea here is that different heuristic functions perform better in different regions of the state space. This variation of *Fast Downward* uses two heuristic functions, the other one being the heuristic used by *FF*.

3. Focused Iterative Broadening Search

Iterative broadening search introduced by Ginsberg and Harvey (1992) is an approach in which a search algorithm only considers b options at each state, increasing the value of b iteratively till a solution is found. In the variation used in *Fast Downward*, the algorithm focuses on one goal at a time, restricting its attention to operators needed for that goal. This is called the *reach-one-goal* approach. In addition, the method forbids certain operators that might undo certain goals. However, as is well known, it is not possible to serialize a set of goals always, and therefore the strategy becomes incomplete.

As described in (Helmert, 2006), *Fast Downward* performs better than *FF* in many domains. The interested reader is referred to (Helmert, 2004; 2006; 2008) for a more detailed study of the approach to using causal graphs.

10.5 Durative Actions

The planning operators specified in the simplest domains (STRIPS/PDDL/ADL) do not deal with a notion of time. There is only a sequencing of actions. Actions can happen in parallel, but then they happen together, at the same instant. The real world actions that are being modelled have durations, but the planner treats them as being instantaneous. The language PDDL2.1 (Fox and Long, 2003) introduces

the notion of *duration* for actions. The duration could be static, or determined dynamically during the course of planning. *Durative actions* invoke the notion of time, and planning with durative actions is called *temporal planning*.

Consider a planning task in which one wants to cook dinner. Let us say that the following plan to cook a simple and tasty¹⁷ dinner is found in the STRIPS domain. The high level plan is to cook some *kali dal*, four *chapatis* and one *papad*. Further, let us say that one is using a *Graphplan* like algorithm that will give us a parallel, low level plan. Also, that there are two gas stoves, S_1 and S_2 , on which cooking can be done. Then the following is an optimal plan in the number of time steps,

$$\{ \{KaliDal(S_1), Chapati^1(S_2)\}, \{Chapati^2(S_1), Chapati^3(S_2)\}, \\ \{Chapati^4(S_1), Papad(S_2)\} \}$$

The actions here are conveniently named after what they cook, and superscripts mark different instances of the same operator. Each action has a precondition that the named stove is available, and the effect is to cook the object. The plan contains six actions arranged in three layers. Now consider the situation when each of the actions above has duration. Let the durations in minutes be $\langle KaliDal, 45 \rangle$, $\langle Chapati, 5 \rangle$ and $\langle Papad, 3 \rangle$. Now a property of interest is the *makespan*, or the total time that the plan needs to run to completion. Figure 10.20 shows a plan with the smallest makespan.

The plan in the STRIPS formulation had sets of actions that happened at the same layer or time step. Working with durative actions, it is not straightforward to say that two actions are happening at the same time. For example, both $Chapati^1(S_2)$ and $Chapati^2(S_2)$ are happening at the same time as $KaliDal(S_1)$, but obviously not at the same time as each other.

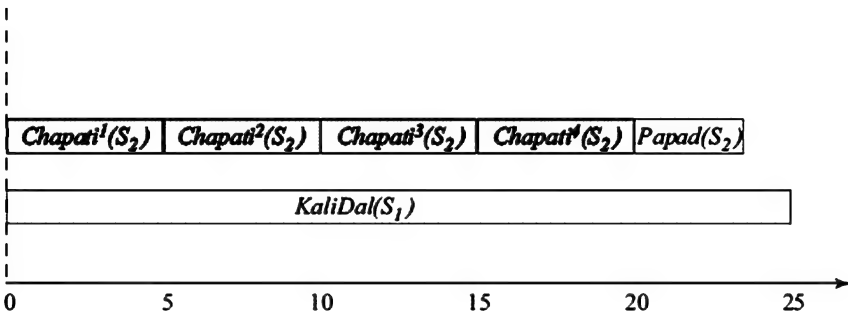


FIGURE 10.20 Plans with durative actions are arranged on a timeline. The notion of two actions happening at the same time no longer holds because they may have different durations.

Two instantaneous actions a and b have only three possible qualitative temporal relations between them—*Before*(a, b) or $a < b$, *SameTime*(a, b) or $a = b$, and *After*(a, b) or $a > b$ —saying respectively that the action a could happen before, at the same time, or after action b . This

is because temporally, the actions are points on an integer line. When we talk of actions with durations then they are represented by intervals over a real or integer line. The relations between two durative actions are described in Allen's *interval algebra* (Allen, 1983), as opposed to point algebra when actions are instantaneous. There are thirteen possible *qualitative relations* between two intervals (durative actions) as shown in Figure 10.21. Qualitative relations do not consider quantitative values such as actual durations of actions or when actions occur. They simply describe how two actions are placed *relative* to each other on the timeline.

When we talk of an action being before (or after) another action, we also have to take into account the case when the later action begins just when the former ends. This distinction was not needed with instantaneous actions because it would correspond to the *SameTime* relation. Therefore, one can see that while the *Before* and *After* relations of point algebra carry over to the interval algebra, the *SameTime* relation gets refined into eleven different relations.

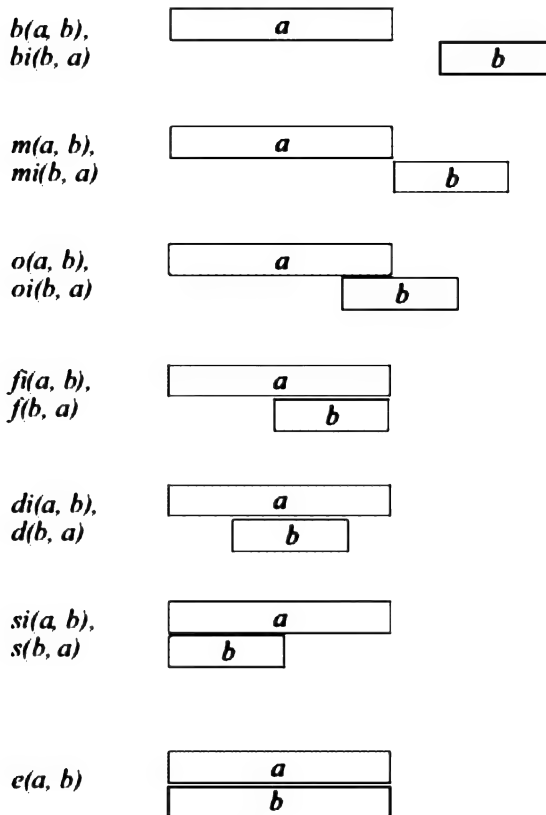


FIGURE 10.21 The 13 relations in Allen's interval algebra. The following six - *b* = before, *m* = meets, *o* = overlaps, *f* = finishes, *d* = during and *s* = starts - have inverses *bi*, *mi*, *oi*, *fi*, *di* and *si*.

The last relation $e = \text{equal}$ is symmetric and does not have an inverse.

The thirteen relations between two intervals, a and b , in Allen's interval algebra are given in Table 10.1. They are also described by the point algebra relations between the end points of the two actions a_{start} , a_{end} , b_{start} , and b_{end} . In addition, with every action a , there is an implicit constraint or relation $a_{start} < a_{end}$.

Table 10.1 The 13 relations in Allen's interval algebra

Name	Description	Inverse	Point algebra description
$b(a, b)$	a is before b	$bi(a, b)$	$a_{end} < b_{start}$
$bi(a, b)$	a is after b (before inverse)	$b(a, b)$	$a_{start} > b_{end}$
$m(a, b)$	a meets b	$mi(a, b)$	$a_{end} = b_{start}$
$mi(a, b)$	a is met by b , (meets inverse)	$m(a, b)$	$a_{start} = b_{end}$
$o(a, b)$	a overlaps b	$oi(a, b)$	$a_{start} < b_{start} < a_{end} < b_{end}$
$oi(a, b)$	a is overlapped by b , (overlaps inverse)	$o(a, b)$	$a_{end} > b_{end} > a_{start} > b_{start}$
$f(a, b)$	a finishes b	$fi(a, b)$	$a_{start} > b_{start}$ and $a_{end} = b_{end}$
$fi(a, b)$	a is finished by b , (finishes inverse)	$f(a, b)$	$a_{start} < b_{start}$ and $a_{end} = b_{end}$
$d(a, b)$	a is during b	$di(a, b)$	$a_{start} > b_{start}$ and $a_{end} < b_{end}$
$di(a, b)$	a contains b , (during inverse)	$d(a, b)$	$a_{start} < b_{start}$ and $a_{end} > b_{end}$
$s(a, b)$	a starts b	$si(a, b)$	$a_{start} = b_{start}$ and $a_{end} < b_{end}$
$si(a, b)$	a is started by b , (starts inverse)	$s(a, b)$	$a_{start} = b_{start}$ and $a_{end} > b_{end}$
$e(a, b)$	a is equal to b	$e(a, b)$	$a_{start} = b_{start}$ and $a_{end} = b_{end}$

Two actions are independent if they are nonmutex. Independent actions can be executed in parallel. Plans with instantaneous actions can have actions in parallel, only if they are independent.

When we have durative actions, apart from independent actions happening in parallel, there may be situations when actions are *required* to be executed in parallel in one or more specific relation from the interval algebra. The term *required concurrency* was introduced in (Cushing et al., 2007). These requirements are not stated explicitly, but are implied by the preconditions and effects of durative actions. Durative actions can be described in PDDL2.1. The following is a simplified description of the (cook) *Chapati* operator.

```

(:durative action chapati
:parameters (?c - chapatiDough ?s - stove ?g -ghee)
:duration ( = ?duration 5)
:condition (and (at start (rolled ?c))
                (at start (free ?s))
                (at start (lighted ?s))
                (overall (lighted ?s))
                (at end (applied ghee)))
:effect (and (at start (not (free ?s)))
             (at start (cooking-on ?c ?s))
             (at end (cooked ?c))
             (at end (not (cooking-on ?c ?s)))
             (at end (free ?s)))
)

```

Durative actions in PDDL2.1 have the following characteristics.

- Preconditions can be at the start of the action, at the end of the action and during the entire action, marked by “*at-start*”, “*at-end*” and “*overall*” before the fluents. The *overall* conditions are over the open interval defined by the end points of the action.
- Effects can be either at the start or at the end, marked by “*at-start*” and “*at-end*”. The effects are themselves instantaneous, happening either at the beginning of the action or at the end. The language does allow us to model continuous metric effects using the parameter *#t* to represent time since the start of the action. If *fuel-level* is a metric fluent then one can say (decrease (fuel-level ? vehicle) (* #t (consumption-rate ?vehicle))) in the effects. Continuous effects can be evaluated at *any time* during the interval of the duration of the action.

In the example for the *Chapati* action, the preconditions are that a stove should be free at the start and the dough rolled, the *overall* conditions are that the stove should be lit throughout the action duration, and the *at-end* condition is that ghee has been applied (on the *chapati* that is cooked¹⁸). The *at-start* effect of this action is that the stove is no longer free (which means that only one *chapati* can be made at a time on it). The stove is again made free at the end of the *Chapati* action, along with the *chapati* being in a cooked state.

Another example of a durative action is when one drives a car from point *A* to point *B*. The action needs the following conditions—at the start the agent should be at point *A*, a car should be available, and the amount of petrol in the car should be greater than a certain amount which is a function of the distance and the rate of consumption. The *overall* condition is that the agent should be in the car and driving it, and the *at-end* condition is that there should be parking space at point *B*. The *at-start* effect is that the agent is not at point *A*, nor is the car, and that the

agent is in the car and driving it. The *at-end* effect is that both the agent and the car are at point *B*. The reader is encouraged to express the above action in PDDL2.1.

At-end conditions in durative actions imply that an action that a planner considers, because the *at-start* conditions are met, may in fact not be feasible if its *at-end* conditions are not met. This implies that a search algorithm has to take care of withdrawing the *at-start* effects, and their consequences if any, if an action becomes infeasible due to unsatisfied end conditions.

Looking at the role of ghee in garnishing a *chapati*, we can introduce a new action *SpreadGhee* which will spread the ghee on the chapati being cooked. The ghee must be spread before the *Chapati* action ends and the spreading action must not end before the the cooking action ends. Given that the duration of *SpreadGhee* is about half a minute, we can see that we are trying to express the following constraint between the two actions¹⁹,

oi(SpreadGhee, Chapati)

That is, *SpreadGhee* must be *overlapped* by *Chapati*. The *SpreadGhee* operator can be defined as follows,

```
(:durative action spreadGhee
:parameters (?c - chapatiDough ?s - stove ?g -ghee)
:duration ( = ?duration 0.5)
:condition (and (at start (cooking-on ?c ?s))
                (at start (available ?g))
                (at end (cooked ?c))
:effect      (at start (applied ghee))
)
```

This action has the *at-start* condition that the *chapati* must be cooking on the stove and it has an *at-start* effect that ghee is applied. This *at-start* effect produces the *at-end* condition for the *Chapati* action. The *Chapati* action has an *at-end* effect that is the *chapati* is cooked, which is an *at-end* condition for the *SpreadGhee* action. As we can see, one can successfully carry out a (making) *Chapati* operation, only if one also begins the *SpreadGhee* action before the *chapati* has finished cooking. In turn, the *SpreadGhee* action can be successfully applied, only if the action ends when the *chapati* making action has already completed. This illustrates a situation in which we have two actions that are not independent, and are in fact closely dependent on each other—generating preconditions for each other at different time points. The actions *have* to be done in parallel, but only in a specific way described by the above constraint on their durations²⁰. The relation between the two actions is shown in Figure 10.22 below.

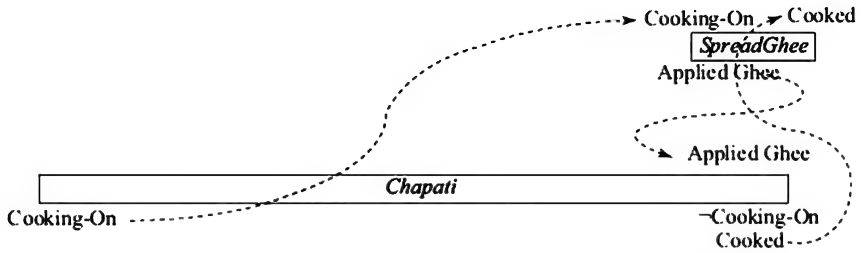


FIGURE 10.22 The required concurrency for the two actions *Chapati* and *SpreadGhee* action. *SpreadGhee* must start during the *Chapati* action, and must end after the *Chapati* action. The dashed arrows show the causal links of producing and consuming fluents.

One can imagine other problems where the required concurrency demands a different set of relations. For example, in many situations, an action might make a fluent true only during its duration. This can be done by producing it at the start and deleting it at the end (as illustrated in the *Chapati* action for fluent *cooking-on*). There could be other actions that require this fluent as an overall condition. For example, a *LightMatch* action may produce the fluent *light*, which might be needed for searching for something or the *OperateCoalMine* action described in (Coles at al., 2008) that keeps a mine operational that is needed for the *MineForCoal* action. Concurrency may also be required for a particular planning problem when one is to plan with deadlines. For example, if one is to cook twenty *chapatis* in the cooking problem then it may be necessary to cook them two at a time on the two stoves. In other words, exploiting concurrency can result in plans with the shortest makespan, and may be necessary if there is a bound on the allowed makespan.

It is the required parallelism that motivates us to look for new planning algorithms. If one were interested in plans that were sequential, then one could simply collapse the durative actions into instantaneous ones, introduce duration as a metric fluent, and look for an optimal plan using the methods described earlier.

10.5.1 Temporal Planning Graphs

In *Temporal Graphplan* (TGP), Smith and Weld (1999) adapted the *Graphplan* algorithm to plan with a simplified form of durative actions. They assume that (1) all preconditions hold at the start, (2) preconditions not affected by the action itself hold throughout its execution, and (3) effects are defined only at the end of the action. The bi-level planning graph constructed by *TGP* avoids duplication of multiple layer representation in the manner described above for *STAN*. As Smith and Weld argue, a bi-level graph makes even more sense for temporal planning, since there is no intrinsic notion of sequencing of layers, because different actions can have different durations. The labels for actions and propositions are not layer numbers but real values indicating start times. Given that different actions executing at the same time may

span different time intervals, it also introduces a notion of an action-proposition mutex in addition to an action-action and proposition-proposition mutexes of Graphplan. However, for the durative actions as defined in PDDL2.1, the mutex definition can be extended to identify more mutex relations. A new definition for mutexes was used in a system called *TPSYS* (Garrido et al., 2002). Two durative actions a and b in *TPSYS* are statically mutex, if one of the following four conditions holds.

1. **AA_{start-start}** Actions a and b cannot start together, if their *at-start* conditions are contradictory or an *at-start* effect of one conflicts with an *at-start* effect of the other. That is, relations $s(a, b)$, $si(a, b)$ and $e(a, b)$ are disallowed.
2. **AA_{end-end}** Actions a and b cannot end together, if their *at-end* conditions are contradictory or an *at-end* effect of one conflicts with an *at-end* effect of the other. That is, $f(a, b)$ or $fi(a, b)$ or $e(a, b)$ is disallowed.
3. **AA_{start-end}** Action b cannot begin when action a ends if an *at-end* effects of action a conflicts with an *at-start* condition or effect of b . In this case, $m(a, b)$ is disallowed. If *TPSYS* needs to schedule an action just after another one, it inserts an infinitesimal gap between them to make sure that the negative interaction does not happen.
4. Action a cannot start or end during the execution of action b , if any effect of action a conflicts with an *overall* condition of action b . For the *at-start* effect of a conflicting the relations $d(a, b)$, $f(a, b)$, and $oi(a, b)$ and disallowed, and similarly for the *at-end* effect $o(a, b)$, $s(a, b)$ and $d(a, b)$ are not allowed.

The static proposition-proposition mutexes depend upon the action-action mutexes as before. The static proposition-action mutex is defined as follows:

5. **PA** A proposition p is mutex with an action a if it is deleted by an *at-start* or *at-end* effect of action a .

TPSYS constructs a planning graph in which each layer $A_{[t]}$ or $P_{[t]}$ is labelled with time t that marks the beginning or the end of an action or proposition respectively. As new actions and propositions are introduced, dynamic mutex relations arise which may disappear in future layers. The following are a few examples of dynamic mutexes that can be defined. The reader is referred to (Garrido et al., 2002) for more details.

AA_{[t]start-start} Two actions a and b are *start-start* dynamic mutex at time t , if one of the following holds.

- Actions a and b are **AA_{start-start}** mutex.
- Some *at-start* condition of action a is mutex with some *at-start* condition of action b .

PA_{[t]start-start} For every action b_i that produces proposition p at time t , an action a and p are *start-start* dynamic mutex at time t , if one of the following holds.

- Actions a and b are $AA_{\text{start-start}}$ mutex
- Some *at-start* condition of action a is mutex, with some *at-start* condition of action b .

$AA_{[t]_{\text{end-end}}}$ Two actions a and b are *end-end* dynamic mutex at time t , if one of the following holds.

- Actions a and b are $AA_{\text{end-end}}$ mutex
- Some *at-end* condition of action a is mutex with some *at-end* condition of action b .
- The two actions could not have been started together at the time when the later of the two began. That is, if D_a and D_b are their respective durations then actions a and b are $AA_{[t - \min(D_a, D_b)]_{\text{start-start}}}$

Research in planning with durative actions and metric resources shifted to state space search subsequently. We describe below two well-known programs exemplifying two approaches. The program *Sapa* adapts the algorithm A^* to search with durative actions and metric resources directly. The program *CRlKEY3* splits each durative action into two instantaneous ones and deploys an *FF* like approach for forward state-space search. Both algorithms use a variation of a relaxed temporal planning graph to guide search, which becomes a critical feature given that the spaces they explore are much larger than the ones encountered by planning in the STRIPS domain.

10.5.2 *Sapa*

Sapa (Do and Kambhampati, 2001a; 2003) is a forward state space search metric temporal planner that can handle deadlines on goal achievement. The search space defined by *Sapa* is made up of nodes of the following kind. Each node is a tuple $S = (P, M, \Pi, Q, t)$ where,

- t is the time stamp of the current state,
- $P = \{ \langle p_i, t_i \rangle \mid t_i < t \}$ is a set of logical fluents p_i true at time t , and t_i is the time at which t_i was last made true,
- M is the set of values for all metric fluents at time t ,
- π is a set of *overall* conditions that need to be preserved at time t , and
- $Q = \{ \langle m, q_i, t_i \rangle \mid t_i > t \}$ is an event queue of all events scheduled to happen after time t , along with the times they are scheduled to occur. m is the mode stating what kind of change is to happen. These events are the *at-end* effects²¹ of actions that have been selected, and the *overall* conditions that need no longer be protected and can be removed from P .

In addition, one needs a structure to store the plan being synthesized. The search algorithm in *Sapa* has two kinds of moves. The first kind selects and adds a new action to the partial plan, and the second kind

advances the time to the next event in the event queue. The latter is applicable when the event queue is not empty. Both kinds of moves modify the state S . An action a is applicable in a node $S = (P, M, \Pi, Q, t)$, if the following conditions hold.

- All logical preconditions of the action are satisfied in P .
- All metric preconditions are satisfied in M .
- No effect of the action interferes with any persistent condition in Π or a future event in Q .
- There is no event in Q that interferes with the *overall* conditions of action a .

Interference is defined as any of the following conditions.

- An action a adds an event e that causes fluent p and there is another event e' in Q that causes $\neg p$.
- Condition p is protected in P till time t_p and action a deletes it before that time.
- An action a has an *overall* condition p and there is an event in Q that deletes p , while a is executing.
- Action a changes, or accesses, the value of a metric fluent which is being accessed by another executing action.

When an action is selected, the following changes are made:

- Its *at-start* effects are added to F and M as the case may be.
- Its *overall* conditions are added to Π .
- Its *at-end* effects are added to the events queue Q with time stamps derived from its duration.

The *AdvanceTime* move can be selected if the event queue is not empty. If it is selected, then the following changes are made.

- The current time t is advanced to the earliest time at which an event is scheduled in Q .
- All the events in Q that are to happen at this time are incorporated into F , M and P appropriately. This may include adding or deleting logical fluents to F , modifying values in M , and deleting persistence requirements from Π .

Figure 10.23 illustrates the state space explored by *Sapa*. It is interesting to note that *Sapa* can schedule actions in parallel, even though the search algorithm picks one action at a time. Once a durative action is selected, its *at-start* effects are added to F and M , and its *at-end* effects are inserted into Q . Till the time stamp advances to the *at-end* effects, the action can be thought of as “being executed”.

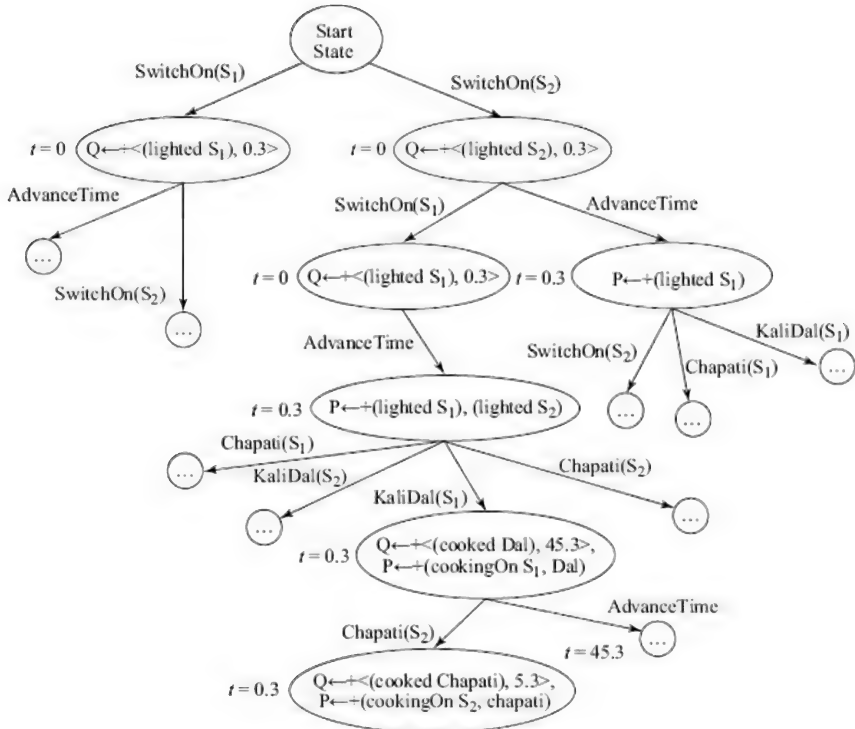


FIGURE 10.23 The search space explored by *Sapa*. The “ $\leftarrow+$ ” operator augments the concerned list.

After making this move, *Sapa* is in a new node its search space. In this node, it is free to choose any move. If it selects a new action to be added then this new actions is in parallel with the one it selected earlier, since the time stamp is still the same. We illustrate this with the planning task depicted in Figure 10.19, along with the additional operator of switching on the stove with duration 0.3 minutes. In the figure, only relevant additions to the search state are depicted. As one can see, in the start state it has two possible moves, switch S_1 on or switch S_2 on. Whichever it selects, it will modify the event queue Q by adding *Lighted*(S_1) or *Lighted*(S_1) as the case may be. Let us look at the right branch, in which it switches S_2 on. In the new search node it has two options, either switch S_1 on or advance time (since Q is not empty). If it chooses the former, shown as the left branch, it reaches a state in which it can only advance time. Once it selects that move, both the stoves are lighted at the same time, 0.3 minutes after start. The two actions have been done in parallel. If it had chosen the latter (right branch) then it would have ended up lighting the two stoves sequentially (if the stove S_1 is lit at all).

Having both stoves lit, it can now start the cooking actions, shown as the four options in the resulting node. The Figure shows a path in which *kali dal* is being cooked on stove S_1 and *chapati* on S_2 , and the time

stamp is 0.3.

The lowermost node in the figure represents a state in the search space in which there are two events in the queue Q . One adds the fluent asserting that the *kali dal* is cooked, and the other, likewise for the *chapati*. The action *AdvanceTime*, if applied now, will advance the time by 5 minutes, the smaller of the two durations in the queue, to 5.3. In the resulting node, the *chapati* will be cooked, the stove S_2 will be free, the *dal* is cooking on S_1 , and the event queue still has the “cooked *dal*” event with the time stamp 45.3. *Sapa* is now ready to cook the second *chapati* on S_2 .

The goals are represented as a set of tuples $G = \{ \langle p_1, t_1 \rangle, \langle p_2, t_2 \rangle, \dots, \langle p_n, t_n \rangle \}$ where the i^{th} goal p_i has to be achieved by the deadline t_i . A state $S = (P, M, \Pi, Q, t)$ satisfies a goal G if for every $\langle g, t_g \rangle \in G$ one of the following holds,

- There exists $\langle p_i, t_i \rangle \in P$, such that $g = p_i$ and $t_i < t_g$, and there is no event in Q that deletes p_i .
- There exists $\langle q_i, m, t_i \rangle \in Q$ that adds $g = q_i$ at time $t_i < t_g$ and there is no event in Q that deletes it.

The search algorithm in *Sapa* is the A^* algorithm (see Chapter 5). Like the heuristic search planners we have seen earlier in this chapter, *Sapa* also uses the notion of relaxation of the problem to arrive at an estimate of the cost involved. In fact, it uses phased relaxation to generate different heuristic functions with a decreasing amount of information gained.

Sapa uses a relaxed version of a two level, temporal planning graph. One must keep in mind that a good, or optimal, solution in metric temporal planning could be defined in various ways. One could count the number of actions, or measure the total time the plan needs to execute (or makespan), or minimize the amount of slack in the plan, or minimize the amount of some resources used. In practice, one may want to optimize on a combination of these criteria. This would also influence the design of the heuristic function, which should be faithful to the criterion being used.

Sapa relaxes the actions in two ways. One, the delete effects of actions are ignored. Two, actions are assumed to not consume resources. That is, the metric fluents in the actions are not decremented in its effects. In either case, the resulting planning graph does not have any mutex relations. The relaxed temporal graph (RTG) uses the following data structures.

A Fact Level A fact f is marked *in* at time instant t_f if it can be achieved at that time instant.

An Action Level An action a is marked *in* at time instant t_a if it can be executed at that time instant.

A Queue of Unexecuted Events Observe that there is one queue for the entire planning graph.

When the construction of the *RTG* begins for a given state $S = (P, M, \Pi, Q, t)$, the fact level contains all the fluents in P marked *in* at time t , the action level is empty, and the event queue contains all unexecuted events of Q that *add* fluents.

An action a is marked *in*, if it is not already marked *in* and all its preconditions are marked *in*. When an action is marked *in*, all its *at-start* effects are marked *in* and all its *at-end* effects are added to the event queue, if not already marked *in* in the fact layer and there is no event in the queue that will add them earlier. After all actions applicable at that time are dealt with, the algorithm returns *no-solution* if either (1) the event queue is empty, or (2) there is an unmarked goal with a deadline earlier than the earliest event in the event queue. This is because now time will be advanced to an instant beyond that deadline. If the event queue is empty, nothing more can be done. If the conditions for *no-solution* do not occur, the *RTG* is extended by advancing the time to the earliest event in its queue and the process continues till all goals are marked *in*. Observe that if at any point the algorithm returns *no-solution*, *Sapa* can abandon that node S and search elsewhere.

When a relaxed temporal graph is returned, it can be used to devise various heuristic functions. The reader should keep in mind our discussion in Chapter 5 on the benefits of having more informed heuristic functions.

If one wants to optimize the makespan of the plan found then one can use an admissible heuristic called *max-span* heuristic, which returns the duration it takes for the last goal to be achieved in the *RTG*. If one is not concerned with *guaranteeing* an optimal solution, then one could use a non-admissible heuristic function, *sum-span*, that adds up the durations for each goal in the *RTG*.

Another measure that can be used to estimate the heuristic function is to measure the *slack* in the *RTG*, where slack is the difference in time when a goal was achieved and its deadline. All three of the heuristic functions that one can define, *min-slack*, *max-slack*, and *sum-slack*, that measure the minimum, the maximum and sum of the individual goal slacks are admissible with respect to the corresponding objective functions.

The *max-span* and the slack based heuristics are concerned with makespan—the total time required to execute the plan. The number of actions and their costs could also be relevant in evaluating a plan. To estimate these costs, *Sapa* extracts a relaxed plan from the *RTG* in a manner similar to the way a relaxed plan is extracted in *FF*. One can then define the *sum-action* heuristic that counts the number of actions in the relaxed plan, and the *sum-duration* heuristic that sums of their durations. However, both these heuristic functions are not admissible.

When one is planning in a domain with metric resources then often actions that augment these resources have to be added in a plan. For

example, a fuelling action is meant solely to increase the fuel level of a vehicle to some desirable level. However, since relaxation ignores reductions in metric resources, the heuristic function may also ignore such actions leading to a less informed heuristic. While it is difficult to account for resource augmenting actions accurately since they depend upon actual values of resources in given states, *Sapa* does an *adjustment*, described below, of the heuristic value based on possible resource consumption.

First, the problem specifications are pre-processed to find actions A_R that increase the amount of resource R maximally by D_R . Let a cost $C(A_R)$ be associated with action A_R , and let $Init(R)$ be level of the resource in the given state S . Given the relaxed plan extracted from the *RTG*, let $Con(R)$ and $Pro(R)$ be the total production and consumption of resource R by the original versions of *all* the actions in the relaxed plan. If $Con(R) > Init(R) + Pro(R)$ then an appropriate number of actions A_R are added to make up for the difference. Let this number be N . Adding N to the count gives us an *adjusted sum-action* heuristic, and adding $N * C(A_R)$ gives us an *adjusted sum-duration* heuristic. For some more improvements to the heuristic function, the reader is referred to (Do and Kambhampati, 2002).

However, this approach to temporal planning has a fundamental limitation. For *Sapa*, time moves from one event to the next (in the event queue), and those are the only time points available to *Sapa* for inserting new actions. It can only place actions in the plan at these time points, and cannot start new actions at other times. The example of required parallelism illustrated in Figure 10.21 cannot be solved by *Sapa*, because it has to start the *SpreadGhee* action a little before the end of the *Chapati* action, and the earliest it can think of after the start of the *Chapati* action is when the *Chapati* action ends.

10.5.3 CRIKEY3

CRIKEY3 (Coles et al., 2008) is built upon *CRIKEY* (Coles et al., 2009) which splits the durative action into two instantaneous actions, called *snap actions*. One snap action happens at the start of the durative action, and the other at the end. Given an action a , the two snap actions are denoted by a_+ and a_- . The start effects of the action (which is the same as the effects of the start action) happen after an infinitesimal duration ϵ , and likewise the end effects. *CRIKEY* searches like *FF* using a relaxed planning graph heuristic.

CRIKEY used a notion of envelopes that kept track of interactions that a started durative action may have with other actions. An envelope is defined by the start action associated with each durative action when it is added, and identifies the corresponding end action. Actions which interact with any existing actions in the envelope are added to the envelope, and in the process may extend the duration and end point of the envelope. A

Simple Temporal Network (STN) is associated with each envelope recording temporal relationships. All such *STNs* have to be temporally consistent in a valid plan. Given an action represented by its snap actions a_{-} to a_{+} , the *STN* has a positive edge from a_{-} to a_{+} with weight equal to its maximum duration and a negative edge from a_{-} to a_{+} , with the weight being the minimum duration of the action. The other temporal constraints are introduced by the planning process, based on preconditions and effects, and the mutex relations. A *STN* is consistent, if there is no cycle with total negative sum of weights.

CRIKEY3 uses an alternative representation of state, and employs only one *STN* for the entire plan. A node in its search space is a triple $S = \langle F, E, T \rangle$ where F is a state described by the set of fluents (including both propositions that are true and values of metric fluents), E is an ordered list of events recording actions that have been started in F but not ended, and T is the set of temporal constraints over the partial plan that has been constructed so far, forming the *STN*. Each entry $e \in E$ is a tuple $\langle op, i, dmin, dmax \rangle$ made up of four elements where,

- op is the identifier of the associated start snap action a_{-} ,
- i is the index of the above action in the plan being constructed, and
- $dmin$ and $dmax$ are respectively the minimum and maximum duration of the associated durative action. If the duration is static then the two values will be equal, but if the duration is dynamic, then the bounds are determined during planning.

CRIKEY3 does *FF* like forward state space search over the snap actions. An action a (it could be a start action or an end action) is applicable in a state $S = \langle F, E, T \rangle$ if,

- F satisfies the preconditions of the action a .
- Action a does not delete any *overall* conditions of the actions present in E .

When the action applied is a start action then a new element is added to E , along with changes in F that are the effects of the action to give us a successor state $\langle F', E', T \rangle$. When an end action is applied, there may be more than one start action it could be paired with. This results in more than one possible successor state, along with an appropriate temporal constraint on the duration of the composed action.

CRIKEY3 can also handle *Timed Initial Literals (TIL)* which are fluents that are controlled by the external environment becoming true or false at predetermined time points. For example, a shop may open at a 10 a.m. and close at 8 p.m., an eclipse may occur during a particular time interval, or a Low Earth Satellite may be visible to a ground station, only during a specific time window (see (Kavuleri and Khemani, 2004) for an extension of *Sapa* to handle TILs). Let t_s be the time stamp associated with a *TIL* p .

CRIKEY3 handles TILs by using dummy TIL actions at each point of time and incorporates p into F if it does not violate any *overall* condition of any action in E . A temporal constraint of the form $\{t_s < t(i) - t(a_0) \leq t_s\}$ is also added, where a_0 is a dummy start action.

Given a partial plan, *CRIKEY3* imposes a total order on the actions and eliminates the *meet* relation between two actions and introducing a minimal separation ϵ between two actions. But instead of constructing envelopes and using an *STN* for each envelope like *CRIKEY* does, it does some advance reasoning that can detect some of the cycles in *STN* before they occur.

The state representation is augmented with additional information being stored with each event $e_{-} \in E$. This additional information constitutes of two values $tmin$ and $tmax$, representing the minimum and maximum time that could have elapsed since that event. These values are updated whenever a successor state is generated, advancing the time forward. If the new state is generated by adding a start action to the plan, time moves forward by an amount ϵ , incrementing both $tmin$ and $tmax$ values for elements of E . If an end event x_{-} associated with an element $x_{-} \in E$ then,

- for every $y_{-} \in E$ that precedes x_{-} in the total ordering, $tmin$ is increased by $\max[x_{-}.dmin - x_{-}.tmax, \epsilon]$ and $tmax$ by $\max[x_{-}.dmin - x_{-}.tmin, \epsilon]$, and
- for every $y_{-} \in E$ that comes later than x_{-} in the total ordering, $tmin$ is incremented by ϵ , and $tmax$ is increased by $\max[x_{-}.dmin - x_{-}.tmax, \epsilon]$.

Then, for any new state generated in search, if for some event $e_{-} \in E$ it is the case that $tmin \geq dmax$ then that state can be pruned, because the durative action e clearly cannot happen because more time than its maximum duration has already passed. If search had continued till e_{-} was added then at stage the *STN* would have had a (negative overall weight) cycle.

This is illustrated in Figure 10.24 with the following example from (Coles et al., 2008). Let us say that we have two durative actions, a and b , with static durations, such that the duration of b is larger than that of a . Since durations are static, it means that for each action, the values $dmin$ and $dmax$ are the same value. Let the planner introduce two snap actions a_{-} and b_{-} , and is about to add action b_{-} resulting in the temporal relations shown in the figure. At the point when action b_{-} is added, the time elapsed $a_{-}.tmin$ since a_{-} is $(b_{-}.dmin + \epsilon)$. Since this is greater than its maximum possible duration, $a_{-}.dmax$ *CRIKEY3* can prune this state itself during search. If the search algorithm had gone on and added a_{-} next then the *STN* made up of the four snap actions would have been inconsistent. Ignoring the two ϵ duration constraints, the sum of the weights in the cycle shown in the figure would be negative.

CRIKEY3 uses a forward state space search algorithm similar to *FF*. It uses a relaxed temporal planning graph to guide search, and like *FF* uses the notion of helpful actions. However, since the instantaneous snap actions are not independent, but come in pairs, it uses a modified relaxed temporal planning graph for estimating the heuristic value of a node. The end actions can only be applied *after* the corresponding start actions have been. This is achieved by adding a dummy precondition to the end actions, and generating it by the corresponding start action.

The interesting point about *CRIKEY3* is that it tends to separate the decisions concerning *which* actions to choose from *when* to schedule those actions. For every start action e_+ the corresponding end action e_- can *only* be applied *after* its earliest possible completion. As long as this is done, the actual scheduling of actions can be done by a scheduler after a candidate plan has been found. Reasoning with the snap actions enables us to pick the required actions, as illustrated by the following example.

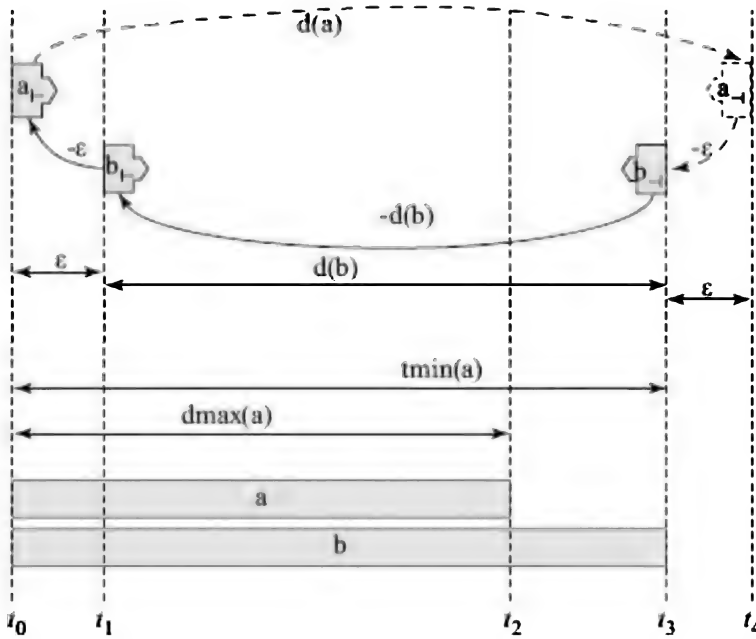


FIGURE 10.24 The state consistency check used by *CRIKEY3*. Actions a_+ , b_+ , and b_- have been added to the plan. The earliest action a_+ can be scheduled now is time point t_4 . If added the corresponding, STN would have a negative cycle as shown. *CRIKEY3* can prune this state itself because the minimum time elapsed after action a started is more than its maximum duration.

Consider the *Chapati* and the *SpreadGhee* actions described above with the corresponding snap actions C_+ , C_- , SG_+ , and SG_- . The required concurrency is that the *SpreadGhee* action begins before *Chapati* ends, and ends after *Chapati* ends. This is shown in the top part of Figure 10.25

with only the relevant conditions and effects marked. A planner like *Sapa* that adds complete actions *in toto* to the plan is unable to schedule the *SpreadGhee* action at any of the time points it reasons with. The time points available to it are the start times and end times of the *Chapati* action. It cannot schedule *SpreadGhee* at the former because its *at-end* condition will not be met, and neither can it schedule it at the latter time point because its *at-start* condition is not met.

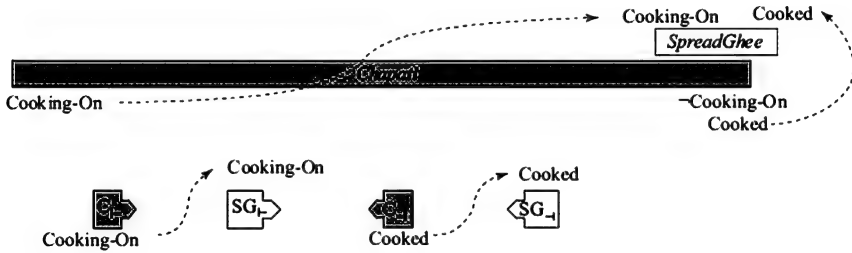


FIGURE 10.25 The required concurrency for the two actions *Chapati* and *SpreadGhee* needs the *SpreadGhee* action to straddle the *end* of the *Chapati* action. Reasoning with snap actions facilitates the interleaving that is required.

CRlKEY3, on the other hand, differs in its approach in two ways. It works with the snap actions separately and secondly, it defers the actual scheduling to a second stage. Because it works with the snap actions independently, it is able to find the correct sequence ($C_- \rightarrow SG_+ \rightarrow C_+ \rightarrow SG_-$) as shown in the lower part of the figure. And because it fixes the actual times later, it is not faced with the problem of when exactly to start the *SpreadGhee* action. The action durations will force the scheduler to schedule it at a time point (in the case $4.5 + \epsilon$) not accessible to a planner that selects and schedules actions at the same time.

This behaviour is in fact reminiscent of the way plan space planning was able to find an optimal plan for the Sussman anomaly by not confining itself to one subgoal $On(A, B)$ or $On(B, C)$ before the other (see Figure 7.13). The algorithm *PSP* (Figure 7.17) was able to separate the concerns of selection and sequencing of actions. In fact, *CRlKEY3* begins by abandoning the total order on actions created during the first phase and reconstructing a partial order based on maintaining causal links between actions including the ordering between start and end actions, and the need to protect conditions from being undone by some actions in a manner similar to plan space planning. The interested reader is referred to (Coles et al., 2009) for more details. A more explicit separation of the sequencing of actions from their selection is implemented in the system *POPF* (Coles et al., 2010).

The algorithm for generating the relaxed planning graph adapted from (Coles et al., 2008) is described in Figure 10.26. The algorithm is designed with the following features, not all of which were present in the relaxed temporal planning graphs used by earlier planners.

```

TemporalRPG ( $S = \langle F, E, T \rangle$ )
1   $f_0 \leftarrow F$ 
2   $t \leftarrow 0$ 
3  for each  $a_-$ 
4      if empty( $\{e \in E \mid e.op = A\}$ )
5          then  $earliest(A) \leftarrow \infty$ 
6          else  $earliest(A) \leftarrow 0$ 
7  while  $t < \infty$ 
8       $f_{t+\epsilon} \leftarrow f_t$ 
9       $actions_t \leftarrow \{a_- \mid precondition(a_-) \subseteq f_t \wedge earliest(A) \leq t\}$ 
10     for each new  $a_- \in actions_t$ 
11          $f_{t+\epsilon} \leftarrow f_{t+\epsilon} \cup add(a_-)$ 
12      $actions_t \leftarrow actions_t \cup \{a_+ \mid precondition(a_+) \subseteq f_t\}$ 
13     for each new  $a_+ \in actions_t$ 
14          $f_{t+\epsilon} \leftarrow f_{t+\epsilon} \cup add(a_+)$ 
15          $earliest(a) \leftarrow \min(earliest(a), t + lb(a))$ 
16     if  $f_t \subset f_{t+\epsilon}$ 
17         then  $t \leftarrow t + \epsilon$ 
18     else  $endpoints \leftarrow \{earliest(A) > t \mid precondition(a_-) \subseteq f_t\}$ 
19         if empty( $endpoints$ )
20             then break
21         else  $t \leftarrow \min(endpoints)$ 
22 return  $R = \langle f_0, \dots, a_0, \dots, a_n \rangle$ 

```

FIGURE 10.26 Algorithm *TemporalRPG* builds a temporal relaxed planning graph.

1. Respect the PDDL2.1 start-end semantics. No action may be left executing in the goal state. In other words, E must be empty at termination. This implies that the construction of the *RTPG* may continue even after the goals appear.
2. Respect relations between start and end actions. The latter can only be applied after the former.
3. Take action durations into account. Insert an end action, only if the time stamp has moved sufficiently for that action to finish.

The resulting *RTPG* ensures that a dead end is never signalled incorrectly, and also takes into account durations of action.

The algorithm accepts as input a state description of the form $S = \langle F, E, T \rangle$ and returns a relaxed planning graph made up of layers of fluents and actions, $\langle f_0 \dots n, actions_0 \dots n \rangle$. We assume that it has access to the set of ground snap actions needed for planning as well as durations of actions.

The algorithm begins by identifying actions that are in the event list E . For each such action $e_- \in E$, it marks the earliest completion time as 0 (lines 3–6). This implies that the corresponding end action e_+ can be added in this state. Further, the end action can be applied only at the earliest time the action can terminate at, represented by the predicate *Earliest*(e). It then proceeds to build the relaxed temporal planning graph in lines 7 to 21. It then identifies all snap actions that are applicable in that layer, and adds their positive effects in the next layer. For the new start

actions it adds, it also updates their earliest ending times (line 15). In lines 16 to 21, it decides whether it needs to increment the time by ϵ or by the duration of the earliest completing action.

10.6 Trajectory Constraints and Preferences

So far the goals that we have posed in the planning problems are constraints on a final state that the plan ends in. The only role that the plan found or the *trajectory* has been in evaluating the plan. We may prefer a plan which finishes the earliest, or a plan that in which the accumulated cost of actions is minimum, or a plan that uses a minimum amount of resources. We have not imposed any requirements on the plan itself. Trajectory constraints are a generalization of the specifying goal requirements on the final state and allow us to specify constraints a trajectory may need to satisfy in a valid plan.

We look at some examples of trajectory constraints. In the cooking domain, one might want that whenever a stove is switched on, it must be switched off sometime. Or every time a refrigerator door is opened, it must be closed within a certain time. A student planning for her examination might be told that every two hours of study must be followed by a break for half an hour. In the blocks world, one might require that there are never more than (say) two blocks on a given block. A trading agent might have to keep a minimum amount of the cash in hand at all times. A carpenter may need to ensure that applying paint should be preceded by applying a primer. An errant employee might be told to report at least once a day to his boss and may have to plan his day accordingly.

Trajectory constraints introduce a new dimension of complexity in planning. In the following discussion, we confine ourselves to trajectory constraints with instantaneous actions for simplicity. The following trajectory constraints have been specified in PDDL3.0 (Gerevini and Long, 2005). The constraints may be specified in the problem description after the goal description in a field marked with “:constraints”. Given a domain D , a plan p , and an initial state I , the plan π generates a trajectory $\langle (S_0, 0), (S_1, t_1), \dots, (S_n, t_n) \rangle$. The plan π is a valid plan, if the trajectory satisfies the goal G .

$$\langle (S_0, 0), (S_1, t_1), \dots, (S_n, t_n) \rangle \models G$$

The goal is a set of goal descriptions $\langle GD \rangle$ and the expression $S \models G$ is read as “ G is true in S ” (see Chapter 11 for more on logical expressions and their semantics. Each goal description $\langle GD \rangle$ is arrived at by possibly applying a modal operator on a goal description. The simplest possible goal description is a fluent. The other goal descriptions may be described in BNF as follows,

$$(at\ end\ \langle GD \rangle) \mid (always\ \langle GD \rangle) \mid (sometime$$

$$\begin{aligned} & \langle GD \rangle \mid (\text{within } \langle \text{num} \rangle \langle GD \rangle) \mid (\text{at-most-once} \\ \langle GD \rangle :: = & \langle GD \rangle \mid (\text{sometime-after } \langle GD \rangle \langle GD \rangle) \mid \\ & (\text{sometime-before } \langle GD \rangle \langle GD \rangle) \mid (\text{always-within} \\ & \langle \text{num} \rangle \langle GD \rangle \langle GD \rangle) \mid (\text{hold-during } \langle \text{num} \rangle \\ & \langle \text{num} \rangle \langle GD \rangle) \mid (\text{hold-after } \langle GD \rangle) \mid \dots \end{aligned}$$

where the “...” includes all existing goal descriptions. Given a plan trajectory $\langle (S_0, 0), (S_1, t_1), \dots, (S_n, t_n) \rangle$ the meaning of the goal descriptors is described in Table 10.2. For illustration, the modal operators are applied on simple goal descriptions ϕ and ψ . A notion of time is used in some operators indicated by $\langle \text{num} \rangle$. This refers to the index of the state in the trajectory, and not absolute time as used in durative actions.

Table 10.2 The semantics of the modal operators

Modal operator	Condition to be satisfied
ϕ	$S_0 \models \phi$
(at-end ϕ)	$S_n \models \phi$
(always ϕ)	$\forall i : 0 \leq i \leq n, S_i \models \phi$
(sometime ϕ)	$\exists i : 0 \leq i \leq n, S_i \models \phi$
(within t ϕ)	$\exists i : 0 \leq i \leq n, S_i \models \phi$ and $t_i \leq t$
(at-most-once ϕ)	$\forall i : 0 \leq i \leq n$, if $S_i \models \phi$ then $\exists j : j \geq i, \forall k : i \leq k \leq j, S_k \models \phi$ and $\forall k : k > j, S_k \models \neg \phi$
(sometime-after ϕ ψ)	$\forall i : \text{if } S_i \models \phi \text{ then } \exists j : i \leq j \leq n, S_j \models \psi$
(sometime-before ϕ ψ)	$\forall i : \text{if } S_i \models \phi \text{ then } \exists j : 0 \leq j < i, S_j \models \psi$
(always-within t ϕ ψ)	$\forall i : \text{if } S_i \models \phi \text{ then } \exists j : i \leq j \leq n, S_j \models \psi \text{ and } t_j - t_i \leq t$
(hold-during t_1 t_2 ϕ)	$\forall i : t_1 \leq i < t_2, S_i \models \phi$
(hold-after t ϕ)	$\forall i : t < i \leq n, S_i \models \phi$

Some examples of constraints expressed in PDDL3.0 are given below.

“Sometime in the plan, block-17 must be on block-29”

(:constraint (sometime (on block-17 block-29)))

“Only green blocks must be placed on block-29”

(:constraint (always (forall (?b - block) (implies (on ?x block-29) (colour ?x green))))))

“Every block must be picked up at least once in the plan.”

(:constraint (always (forall (?b - block) (sometime (holding ?x))))))

We look at each of the above constraints and suggest some ways to adapt the *Graphplan* algorithm to handle them (Garwal, 2006). This approach requires that the planning problem be modified in a pre-processing phase, and also a few changes to the backward search phase of *Graphplan*. One would imagine though that as the planning problem

becomes more and more complex, it would be desirable to adopt methods that reduce different kinds of problems to a common representation, for example constraint satisfaction, for which existing algorithms can be used.

1. The first two constraints φ and (at-end φ) are the STRIPS goals that have to be achieved in the final state.
2. The constraint (always φ) requires that φ be true in all states in the trajectory, and can never be deleted. It can be handled by removing all actions that have φ in their delete list. This can be done in a preprocessing phase. Also, if φ is not true in the initial state, the planner can return failure immediately.
3. The constraint (sometimes φ) requires that there is at least one state in the trajectory in which φ is true. This can be handled by adding a dummy effect δ to all the actions that have φ in their add effects, and adding the constraint (at-end δ). This will ensure that one of the actions is included at least once in the plan.
4. The constraint (within t φ) requires that the fluent φ should appear in the layer number t or earlier. Add a dummy fluent δ to all the actions that have φ in their add effects. Modify the backward search procedure to include δ as a goal when it reaches level t .
Another approach that is applied in the forward phase, and does not require modifying the procedure is as follows. One could include a *CountDown* action that decrements a non-negative counter C_φ (a metric fluent). This action would have a counter with a value greater than 0 as a precondition, and a decremented counter as an effect. One could initialize the counter C_φ to t in the initial state, and start counting down. Exactly one instance of this action will appear as a parallel action in the first t levels in the plan. One could now add a dummy fluent β to the precondition of all actions that add φ and in the initial state, and include δ as an add effect. Also, include (at-end δ) as a goal. Introduce another action to delete β when the counter reaches a value 0.
5. The constraint (at-most-once φ) requires the fluent φ be added at most once in the plan. This means that when it is added once, it should not be possible to add it again. This can be achieved by adding a dummy fluent δ to the initial state, to the precondition list of all actions that add φ , and to the delete list of all those actions. Since δ is deleted by any action that adds φ , no other action that adds φ can be selected since δ is a precondition for that action.
6. The constraint (sometime-after φ ψ) requires that if the fluent φ is added in some state in the trajectory then ψ must be added in a state that is later in the trajectory. Observe that one ψ can account for multiple occurrences of φ . The following modifications can be used to satisfy this constraint. Add a dummy predicate δ to the initial state, as an (at-end δ) goal, to the delete effects of all actions that

add ϕ , and to the add effects of all actions that add ψ .

7. The constraint (sometime-before $\phi \psi$) says that any occurrence of ϕ in a state must follow a ψ in a preceding state. This means that if an action adds ϕ then an earlier action must have added ψ . This can be satisfied easily by adding a dummy fluent δ to the add effects of all actions that add ψ , and as a precondition for all actions that add ϕ .

8. The constraint (always-within $t \phi \psi$) is like (within $t \psi$) except that *every time* ϕ is added to a state then ψ *must* be added within t steps. Add a dummy predicate δ to the initial state, to the delete effects of all actions that add ϕ , and to the add effects of all actions that add ψ . Modify the backward search phase to return failure, if the consecutive states are without δ .

However, an attempt to adapt the forward procedure described in 4 could lead to a problematic situation. One could initialize the counter C_ϕ to t in the action that adds ϕ , and start counting down. One could now add a dummy fluent β to the precondition of all actions that add ψ and in the initial state, and include δ as an add effect. Also, include (at-end δ) as a goal. Introduce another action to delete β when the counter reaches a value 0. However, if ϕ is added more than once, it would cause a problem because multiple instances of the counter will exist but the first one which becomes 0 will result in β getting deleted once for all.

9. The constraint (hold-during $t_1 t_2 \phi$) requires that ϕ be true at level t_1 and remain true at least till level t_2 is reached. This can be achieved by modifying the backward search phase by adding the goal ϕ starting at level $(t_2 - 1)$ till the level t_1 .

First achieve ϕ within t_1 using counter. Add A to add effect of those actions. when counter is zero also add B . If something deletes ϕ , also delete B (?)

10. The constraint (hold-after $t \phi$) says that ϕ must be true in all levels after t . This can be achieved by adding the fluent to the goal set at each level, until level t is reached.

10.6.1 Preferences

So far, the planning problems we have looked at have goals, and trajectory constraints, that have to be satisfied completely for a plan to be valid. If any goal condition is not satisfied, then the plan is not valid. We call these kinds of goals as *strong goals*, or strong *trajectory constraints*. In contrast, we can define planning problems in which we have goals that we would like to satisfy, but we may still accept a plan that fails to satisfy some goal or trajectory constraint. Of course, we would evaluate such a plan as being of lower quality as compared to a plan that satisfies more or all the goals. We call such goals that we are willing to do without if need be, as *soft goals* or *soft constraints* or *preferences*. Preferences are

desired goals and trajectory constraints and some preferences may be preferred more than others.

For example, someone may have the goal of booking a seat in a train and have a preference of booking a window seat. Or someone planning an outing may have a preference of eating in a South Indian restaurant, with Ethiopian being a close second. Office etiquette may require that one knocks before opening a colleague's door. Planning a walk in Chennai one may have a preference to be in the shade at all times. Someone may have a desire (soft trajectory constraint) that at least once in his lifetime he will have a chance to behold the Khangchendzonga.

Each preference can be assigned a weight that determines the cost of violating that preference. The evaluation of a plan may incorporate such costs along with other costs of actions. Observe, that this converts the planning task to an *optimizing problem* in which one could continue to look for better solutions till an optimal one has been found. Different solutions achieve different (soft) goals. This is in contrast to the planning algorithms we have studied so far that are designed to find a least cost solution for the *same* problem represented by strong goals and trajectory constraints. Another difference with earlier approaches is that specifying preferences in a specific problem instance allows a user to optimize on different features at different times. An approach to adapting *Graphplan* for solving preferences may be to first extend the planning graph till all the strong goals are solved then extend the graph further in an attempt to solve any preferences that are not satisfied. The process of extending the plan (with more actions) will terminate when the cost of adding more actions outweighs the cost of leaving preferences unsatisfied.

In PDDL3.0 (Gerevini and Long, 2005), a preference is specified by the statement: (preference [name] <GD>)

The expression (is-violated <name>) takes on a value of the number of distinct preferences of that name that are violated in a given plan. The naming of a preference allows one to associate different *penalties* with the violation of different constraints. An anonymous, or unnamed, preference is assumed to have weight equal to 1 by default. Preferences may also be included as preconditions of actions in which case the number of violations is the number of times the action is selected violating that constraint.

The following is an example of preferences that one may use in planning an evening out. The descriptions shown below follow the rest of the problem specification. The goals expressed as trajectory constraints are that one should be at the beach from 6 to 8, go to a restaurant at some point, preferably a South Indian or an Ethiopian one, for dinner, also preferably go to a mall sometime, and end up at home. In addition, one should try and avoid making phone calls at all points during the outing.

```

(
  (...
    (:constraints
      (and
        (hold-during 6 8 (at beach))
        (at-end (at home))
        (sometime (go restaurant))
        (preference mall (sometime (at mall)))
        (preference south (sometime (eat-at south-indian-restaurant)))
        (preference ethiopian (sometime (eat-at ethiopian-restaurant)))
        (preference quiet (always (not (making phone-call))))
      )
    (:metric minimize (+
      (* 150 (is-violated south))
      (* 130 (is-violated ethiopia))
      (is-violated mall)
      (* 8 (is-violated quiet))))
  )
)

```

The statements in the `:metric` field assign weights to the preferences. Observe that the preferences for the two restaurants have high weights. These are designed to make sure that one of them is chosen. It also means that the other one will still be a penalty. Notice that the penalty for not visiting a mall is comparatively much smaller.

10.7 Planning in the Real World

The planning techniques we have studied so far focus solely on the task of finding a plan for a given goal. We assume that the planning domain and problem are expressed in a well defined language, designed to express the planning problem. The given problem describes the given world completely and states the goal, and the planning task is to find a plan for that goal. We make a closed world assumption that everything that needs to be known about the world is known, and what is not known is irrelevant. We assume that the world is static and does not change while we are planning. We also implicitly assume that the world does not change while the plan is being executed, except by the actions in the plan. We assume that the actions are effected perfectly in the world and the changes actually made are as described in the operators.

The ability to synthesize *and execute* sequences of actions that would achieve some desired goal is an integral part of intelligent behaviour. An autonomous agent, for example a robotic machine, would have to incorporate the ability to plan, but it would also need to go much beyond that. Actions in the real world are not always deterministic and an agent will need to “watch its step” as it goes about acting in the real world. It will need to monitor its actions and ascertain that the world is indeed as it expected it to be at each stage. And if it not then it should be able to modify its plan or generate a new plan before proceeding further. For example, one may have a plan to draw money from the local bank ATM before proceeding for an outing, but the action may fail if the ATM has run out of cash and the agent may have to modify its plan or devise a new one.

A goal, as specified in problems expressed in PDDL, is likely to be one of many goals the agent may have, perhaps at different times. There

may be other goals that arise due to the need to execute the actions of the plan in a domain, sensing the world, and maintaining the health of the agent. For example, in the foreseeable future, one may be able to ask a robotic agent to bring a cup of tea along with the morning newspaper. A command like this to an agent can be seen as setting goals for it to find plans for and execute them autonomously. The *high level plan* may comprise the kind of actions shown in Figure 10.19. However, the *actual* actions that can be effected in the real world may be much more fine grained. In fact, we can already see that there is going to be an hierarchy in which actions are going to be organized. An action at a higher level may correspond to a plan at a lower level, with its own goal. At a higher level, the action, for example, may be to fetch the newspaper. At a lower level it might mean determining (sensing or asking) the location of the newspaper, and generating a path plan to go where it is. Each of the actions at this level, for example “go to the front door”, may serve as a goal for a lower level that may involve locomotion and navigation. These may involve starting certain motors in its body, and other control functions. We can see that plans and actions, like programs and instructions in programming languages, will have hierarchies of representation. The lowest level actions, like the machine code, is the one made of domain level actions.

Apart from the hierarchical levels discussed above, a robotic agent may have to keep track of actions designed to maintain its balance and perhaps poise, satisfy operating constraints (for example, while moving from one place to another keep the arms by the side, unless carrying a desired object), and keep a check on its own energy levels (resource), inserting a charging action if needed.

An agent may have a large number of goals and may be compelled to treat them as soft goals in a dynamic environment. An autonomous agent may also have to cater to new goals that may arise due to unexpected change, or as new, high level commands are received. Further, it will have to manage its own time, making judicious allocations to sensing, deliberating and acting. A significant amount of engineering effort is required to coordinate and control the activities of its many components. We briefly look at an approach to planning in the real world that has been successfully tried out in two domains.

10.7.1 RAX

Space applications are naturally suited for autonomous agents. Imagine a spacecraft hurtling towards Jupiter in 1994 with the task of tracking the comet Shoemaker-Levy 9²². At a distance of over 800 million kilometres from Earth, it would take about 27 seconds for light to travel from Earth to the spacecraft. Which means it would take 27 seconds for an image sent from the spacecraft to reach Earth, and likewise for a command sent from Earth to reach the spacecraft. It is imperative that the spacecraft be able

to respond autonomously to at least some situations. For example, it may need to swerve to avoid a floating piece of rock, or take an image of a passing one.

In 2008, a signal from the Voyager 1 spacecraft in the outer reaches of the solar system took 14 hours and 52 minutes to reach NASA's Deep Space Network (Poon, 2010). Clearly, controlling Voyager 1 remotely is out of the question. Closer home, one may need autonomy if one had a low earth orbit satellite because it would be visible to a ground station only for a small duration in its orbit, or for a deep sea exploration vehicle. An autonomous system should be able to accept high level commands from its owner, generate plans to achieve the desired goals, execute and monitor the plans, and report back to its owner. The system *MEXAR* (*Mars EXpress ARchitecture*) is an example of a system that is able to plan and schedule its imaging and downloading actions (see for example (Oddi et al., 2002), (Cesta et al., 2007)).

The *Remote Agent* (RA) was a planning and execution system designed to autonomously control NASA's New Millennium Deep Space One aircraft (Muscettola et al., 1998), (Chien et al., 1998)²³. The high level architecture of RA is shown in Figure 10.27 adapted from (Muscettola et al., 1998). The RA communicates with the real time control system that is responsible for sensing the state of the spacecraft and the environment, and controlling the hardware. The real time system also communicates with the ground station. One key feature of the RA architecture was that it produced plans that are *temporally flexible* and which could be adapted to the sensed environment by the executive.

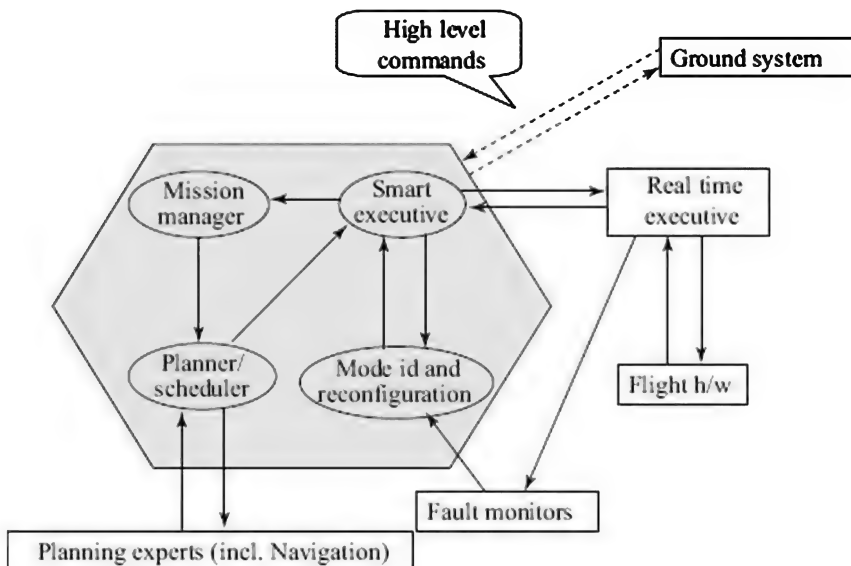


FIGURE 10.27 The Remote Agent architecture.

The RA was made up of four subsystems enclosed in the hexagon in the figure, described briefly below.

Smart Executive(EXEC)

EXEC is a robust, event-driven and goal-oriented multi-threaded execution system. The smart executive is the system that interacts with the real time controller of the spacecraft. It is EXEC that coordinates the activities of multiple subsystems during plan execution and takes advantage of the flexible plan generated to execute it at an opportune time in the planned window. EXEC also monitors the plan execution, exploring alternative ways to achieve the task. When new plans are needed, it describes the state and the goal to the mission manager and requests for a plan. The executive automates the decomposition of goals into smaller activities that can be executed concurrently in different subsystems of the spacecraft. EXEC plays the main coordination role between all flight software modules, both internal and external to the RA. The two main aspects of its behaviour are the following:

- **Periodic planning over extended missions** It periodically asks the PS for new tasks and incorporates the actions into the tasks being executed.
- **Robust plan execution** The responsibility of successfully executing plans in the face of uncertainties and failures lies with the *EXEC*. It exploits the flexibility of the plan supplied by PS to tune it to the existing conditions. The high level algorithm followed by *EXEC* is depicted in Figure 10.28.

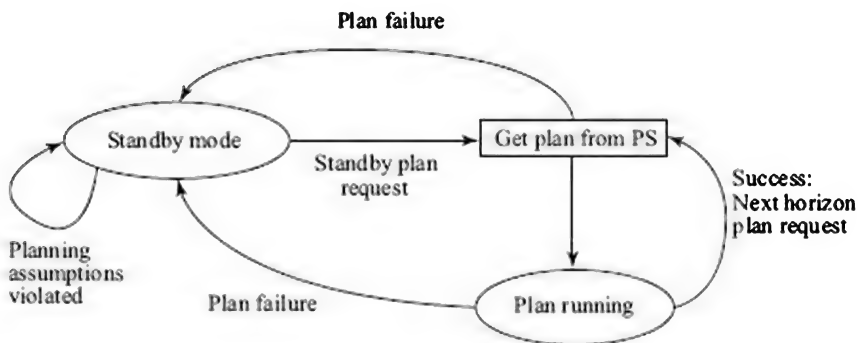


FIGURE 10.28 The smart executive is either executing plans in the current horizon, asking for more plans, or in a standby mode waiting for plans to be repaired.

Mission Manager(MM)

The mission manager is a system that generates short term planning goals, based on the long term mission profile. The Deep Space mission

profile contained long term periodic goals like taking images of objects, and health maintenance goals such as performing an engine calibration activity within a 24-hour window, before approaching the target. The *MM* system accepts goals from *EXEC* and decides upon the full set of goals to be achieved over the next planning window. It then requests the *PS* to generate a plan.

Planner/Scheduler(*PS*)

The *RA* uses a constraint based temporal planner and resource scheduler *PS*. It produces flexible concurrent temporal plans that satisfy trajectory constraints. The plan constrains the activity of each spacecraft subsystem. The *EXEC* takes up each activity as a separate thread and makes the finer grained decisions for actual execution. The planner uses a notion of *timelines* described later to represent different activities. Figure 10.29 illustrates two timelines depicting the fact that the onboard camera can be used to take a picture of an object *B*, only *during* the interval when the camera is pointing towards *B*. The timelines *Camera* and *Attitude* are shown with three intervals, each depicting the activity, or lack of it, for the two attributes of the system.

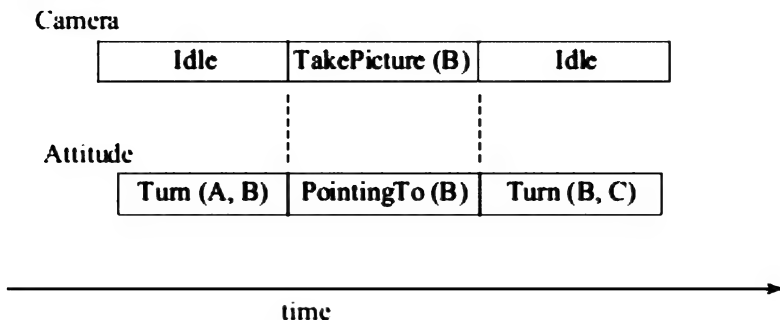


FIGURE 10.29 Timelines in the Remote Agent planner. The Camera timeline describes the status of the camera on board, and Attitude timeline describes the spacecraft orientation. The *TakePicture(B)* action should be *during* the *PointingTo(B)* interval.

The reader would have observed that the timelines do not distinguish between states (e.g. *PointingTo*) and actions (e.g. *Turn*). The *RA* employs the following structural principles in modelling the domain.

- **State Variable Principle** The evolution of the system at any time is described by a set of finite state variables.
- **Token Principle** A single representative primitive called *token* is sufficient to describe the evolution of system state variables over time.

We look at the plan representation in a little more detail later.

Mode Identification and Reconfiguration module(MIR)

The MIR was a discrete, model based controller called *Livingstone* that has a single declarative model of the spacecraft (Williams and Nayak, 1996). The sensing component of MIR, called MI (mode identification), “tracks the most likely spacecraft states by identifying states whose models are consistent with the sensed monitor values and the commands sent to the real time system”. MI reports to *EXEC*, and informs it at a high level, if a sensed state is not what was expected, for example if a thruster has failed. The recovery component, MR (mode reconfiguration), then tries “to find a least cost command sequence that establishes or restores desired functionality by reconfiguring hardware or repairing failed components”. The *EXEC* calls MR with a specification of constraints when it needs to recover from a failure. The MR module is a reactive module, unlike the generative planner, and quickly finds a solution to satisfy the constraints.

MIR is *model based*, in the sense that it works with a single, declarative, compositional model of the spacecraft. For the task of diagnosis, it employed *qualitative models* in which qualitative deviations from normal behaviour were used to identify and isolate faults. The search performed for consistent models is aided by an Incremental Truth Maintenance System (Nayak and Williams, 1997). We look at qualitative, model based diagnosis briefly in Chapter 17. An important conclusion drawn was that it is feasible to use deductive reasoning models in situations where the system is required to respond swiftly. We refer the interested reader to (Muscettola et al., 1998). Here, we simply list the functions performed by MI and MR. The mode identification system does the following in the Remote Agent.

- **Mode Confirmation** Informs the *EXEC* that a particular command was executed successfully.
- **Anomaly Detection** Identifies *observed* spacecraft behaviour that is inconsistent with the *expected* behaviour.
- **Fault Isolation and Tracking** Identifies components whose failure caused the observed anomalies.
- **Token Tracking** Helps plan monitoring by tracking the state of attributes of interest to the executive. The state of the spacecraft is expressed in the form of *configuration goals*. When the spacecraft deviates from active configuration goals, the mode reconfiguration capability finds a least cost set of control actions that moves the spacecraft to an acceptable configuration. MR does this by one of the following.
- **Mode Configuration** Move to a least cost configuration that is acceptable.
- **Recovery** Move the spacecraft from a failure state to one that

restores a desired function, either by repairing a failed component, or by finding alternate ways of achieving those goals.

- **Standby and Safing** If unable to place the spacecraft in a desired configuration, move it to the standby state and wait for the high level planner or the ground station to find a solution.

As reported in (Jónsson et al., 2000), the Remote Agent Experiment (*RAX*) was conducted in May 1999, when the RA controlled the Deep Space One spacecraft completely autonomously. The experiment demonstrates the capability of planning and execution of plans, and also the capability of doing model-based inference for recovery from failures.

In summary, RA achieves robust plan execution (Muscettola et al., 1998) by

- Executing flexible plans by running multiple, parallel threads and using fast constraint propagation algorithms in *EXEC* to exploit plan flexibility
- Choosing a high level of abstraction for planned activities, so as to delegate as many detailed activity decisions as possible to the procedural executive
- Handling execution failures using a combination of robust procedures and deductive repair planning

We next describe some of the features of an autonomous planning system in the context of a deep-sea autonomous vehicle.

10.7.2 T-REX

In January 2004, NASA landed two rovers on the surface of Mars whose operations were controlled by a system called MAPGEN (Mixed-initiative Activity Planning and GENeration) which had the same basic approach of adaptive execution of the RA (Bresina et al., 2005; 2005a). The Mars Exploration Rovers (MERs) were not completely autonomous since they had humans in the loops inspecting and authorizing the plans generated in a mixed initiative manner (Burststein and McDermott, 1996). But like the system for autonomous submarines described here, MAPGEN was based on a system called EUROPA (described later).

The advances in autonomous planning and execution have opened up new vistas in oceanography research. Whereas earlier data from the oceans had to be painstakingly collected by ships zigzagging the ocean surfaces, now such data can be collected by Autonomous Underwater Vehicles (UAV) diving deep into the oceans collecting data, surfacing occasionally to report to their owners.

We look at the system called T-REX (Teleo-Reactive Executive) which is a successor of the RA system we saw above (McGann et al., 2008), (Py et al., 2010). Controlling a UAV requires the system to follow the

sense-deliberate-act cycle of an autonomous agent. In order to scale the operations up, the planning system partitions its scope both functionally and temporally. This implies planning separately for different subsystems and also separating deliberation and action at different time scales. The system is partitioned along the following lines.

- **Functional** Indicating the state variables relevant for deliberation and action. This enables a logical partitioning of different subsystems.
- **Temporal** Indicating the look-ahead window for which planning is done. This separates long term goals from short term reactions to the environment.
- **Timing** The latency allowed for the deliberation process to complete. Some systems may have to react rapidly, whereas others may have more time available for planning. The architecture of T-REX (figure adapted from (McGann et al., 2008)) is shown in Figure 10.30.

It constitutes of four teleo-reactors²⁴ (see (Nilsson, 1994)) communicating with each other and vehicle controller as shown.

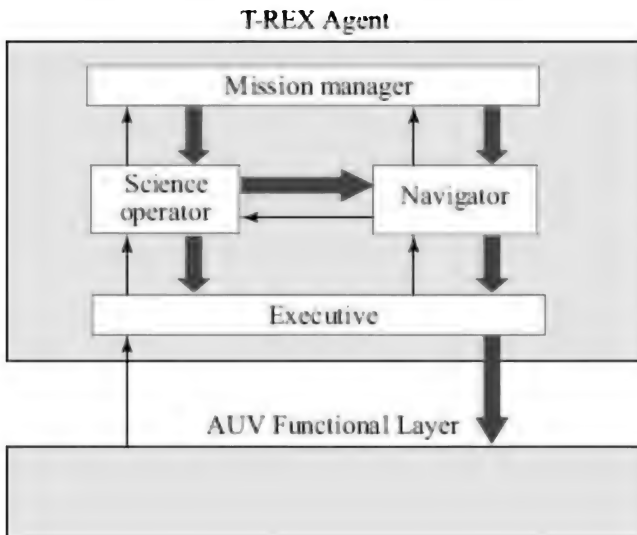


FIGURE 10.30 The T-REX agent is made up of four teleo-reactors and controls the AUV functional layer. Commands flow along the thick arrows and observations flow along the thin arrows.

The Mission Manager is the high level system with the long term scientific and operational goals of the mission. Its temporal scope is the entire mission, and it may take minutes for deliberation. It generates directives for the Science Operator and the Navigator.

The Science Operator refines the high level directives it receives from

the Mission Manager and may also make shorter term decisions based on the science goals. For example, it may instruct the navigator to adopt a particular motion trajectory useful for collecting data. Its temporal scope may be of the order of a minute and its deliberation may be required to be done in the order of a second.

The Navigator is similar to the Science Operator in temporal scope and latency but focuses on the task of navigation. It receives directives from both the Mission Manager and the Science Operator and interprets them for the Executive.

The Executive interfaces the agent with the AUV functional layer. It has close to zero latency, allowing no time for deliberation.

T-REX has an explicit notion of time, measured in *ticks*, that is followed globally by all its subsystems. The state variables, or attributes, are arranged in timelines, like in the RA. Each state variable has a timeline associated with it. A timeline is a series of *tokens* which are temporally qualified assertions, predicates with specifications of start and end times defined as intervals over time. The following features of the timelines are instrumental in interaction between the teleo-reactors.

- **Ownership** Each timeline is owned by exactly one reactor, who decides what goals to instantiate. Other reactors may request new goals of that reactor, or modify their requests if the plan changes. The timeline is *internal* to its owner and *external* to other teleo-reactors.
- **Observations** The owner of the timeline captures the current value of the timeline as observations.
- **Goals** Goals are *desired future values* of the timeline. Goals are requests for refinement into subgoals or commands. A teleo-reactor may recall a goal it had posted earlier.

For example, the Mission Manager may have a science goal for which it needs the UAV to be taken to a certain place. It can specify this to the Navigator by setting a goal in the latter's timeline called *Path*. While dispatching this goal to the Navigator, the Mission Manager needs to make sure that it gives enough time for the Navigator to deliberate and act. On receiving the goal *Go(xLoc, yLoc, depth)* in its *Path* timeline, the Navigator may in turn decide that it needs to dispatch goals *Descend(100)* at tick 10, taking from 50 to 55 ticks and then achieve *Waypoint(xLoc, yLoc)* into the *Command* time line of Executive.

In order to maintain consistency, T-REX needs to ensure that at the end of every tick, the entire set of observations received is consistent with the plan. At every tick, the following algorithm is followed.

1. First, all timelines are synchronized. This may involve different amounts of computation for different timelines. For example, the *Path* timeline owned by the Navigator is not likely change very often, being over a longer duration. The *Position* timeline, owned by

the *Executive*, is likely to change at every tick if the *AUV* is moving. A principle used in the system is that observations dominate expectations. For example, if the Navigator expected the current depth to be a value d_N , but the Executive who owns the timeline determines it to be d_E then the Navigator's plans will need to adjust to the observed depth.

2. After that, all goals are dispatched. These goals should be given to the concerned teleo-reactor so that it can achieve them in the time given. If t is the current tick, l is the latency of the teleo-reactor, and p its planning horizon then the dispatch window for the goal is $[t + \lambda, t + \lambda + \pi]$. As soon as the start time of the goal intersects with this window, it should be dispatched. In the current implementation of *T-REX*, the Executive is assumed to have zero latency, and therefore planning horizon, so the commands are dispatched when they need to be executed.
3. The remaining time is allocated to the teleo-reactors for deliberation. Like the *RA*, *T-REX* too uses a constraint based, temporal planning approach. The variables of the constraint network include state and actions, and the constraints between them are represented as in *EUROPA* (described below).

10.7.3 CAIP/EUROPA

Complex systems like spacecraft, rovers and underwater vehicles are made up of different components; each of which does a different task. The components may constrain each other in various ways, and this interaction needs to be taken care of during planning. A plan is the output of a planning system that influences the behaviour produced by the executive. In a sense, plans determine the behaviour of the system. However, in the context of real world agents, it is important that this behaviour specification by the planner is not precise and rigid. It must leave enough room for the executive to fine tune the behaviour to the actual conditions. The systems described in this section achieve this by representing plans as constraint networks, instead of a sequence of actions that the executive must execute. During execution, the executive interprets the plan represented as a constraint network, choosing consistent values best suited for the actual situation. Once it makes a decision for a value, it needs to propagate constraints (see Chapter 9) to maintain consistency of the plan (network) as execution progresses.

The Constraint based Attribute and Interval Planning (*CAIP*) paradigm (Frank and Jónsson, 2003) is an approach that facilitates the specification of complex planning domains and implementation of planners that can exploit constraint based reasoning. The *CAIP* paradigm builds upon the *RA* planner which was derived from the *HSTS* planner (Muscettola, 1994), and the planner *IxTet* (Laborie and Ghallab, 1995), both of which are based on attributes to represent state variables. The framework has

been implemented in a system called Extensible Universal Remote Operations Planning Architecture (*EUROPA*), the latest at the time of writing being *EUROPA2.1* developed in 2007.

The most important difference between the STRIPS based systems and the attribute based systems is that the latter allow for explicit reasoning about predicates and actions, whereas the former did so implicitly. Consider for example the STRIPS action $Stack(A, B)$ and the resulting fluent $On(A, B)$. The basic search algorithms for planning use actions to generate the search space to explore and the state is represented as a collection of fluents which are true. *Graphplan* and its derivatives introduce the notion of layers, and we can then talk about the layer in which an action occurs or a fluent becomes true or false. Planning as satisfiability or as a CSP extended the action and fluent representations by adding another parameter called time. The approach used in *CAIP* on the other hand, follows the Event Calculus (see Chapter 13) like representation, in which both actions and fluent appear as reified arguments in higher level predicates. For example, the happening²⁵ of a stack action may be represented as,

$$Holds(Arm_1, 10, 10, Stacking(A, B))$$

or

$$Holds(Arm_1, 10, 12, Stacking(A, B)) \text{ if stacking is a durative action.}$$

The statement asserts that during the specified duration, the arm is in a “state of” stacking block A on block B . In this example, Arm_1 is the attribute, and amongst other values, it can take the fluent $Stack(A, B)$ as a value. Other kinds of values it could take are for example *Idle*, or *TuckedIn* (if it is required to be tucked in while the robot moves). Likewise, $Holds(Block(A), 13, 30, On(B))$ could be a description of the fact that the value of the attribute $Block(A)$ during time points 13 and 30 is $On(B)$, an interval during which the block A is on block B . As we saw that in the description of the Remote Agent, both states and actions become values of the corresponding attributes. Thus, Arm_1 can be doing the action of stacking A on B , or it could be in a state described as *Idle*. Both are represented identically using intervals over which the fluent or action is true.

Every such attribute is associated with a *timeline*, and the timeline must account for the attribute at all times during the plan horizon. Thus, state of Arm_1 may be represented as a sequence of intervals, for example ($Hold(Arm_1, 0, 4, Idle)$, $Hold(Arm_1, 5, 9, Moving(A))$, $Hold(Arm_1, 10, 12, Stacking(A, B))$, $Hold(Arm_1, 13, 18, Idle)$). This description is extremely precise. *CAIP* does allow flexibility while inserting such actions (intervals) into the plan. One might assert a lifted version of an interval, for example, $Holds(Arm_1, t_1, t_2, Moving(A))$ where t_1 and t_2 are variables. Then one could add constraints on the values that t_1 and t_2 take. For

example, one could say that t_2 should not be later than some deadline, or that t_1 should begin during a certain time interval. One can also have the value of the attribute as a variable, for example $Holds(LocationRobot_1, 10, t_2, Going(room_1, X))$, which could be used if one wanted the robot to leave $room_1$ without specifying where it goes.

Constraints may be placed on intervals in the same timeline or across different timelines. We have seen an example in Figure 10.28 in which the *PointingTo* interval of *Attitude* must *contain* the *TakePhoto* action interval of the *Camera*. If we have a $LocationRobot_1$ attribute, then we might specify that an interval specifying $Going(Source, Destination)$ must be *met-by* an $At(Source)$ interval, and should *meet* an $At(Destination)$ interval. Representation of such constraints in CAIP is done using the notion of *Compatibilities*.

Compatibilities allow the expression of constraints in the form of rules. For example, if a robot arm has to be tucked in while it is moving, then if $Holds(LocationRobot_1, t_1, t_2, Going(X, Y))$ is in the plan, then $Hold(Arm_1, t_3, t_4, TuckedIn)$ should also be in the plan along with the constraints $t_3 < t_1$ and $t_2 < t_4$. We express the temporal constraints with Allen's interval algebra described earlier. This constraint can be combined with the constraints that the *Going* value must be *met-by* and *meets* two *At* value intervals. The relations between different intervals are captured in *Configurations*. The constraints described here could be expressed as a compatibility rule shown below,

Head: $Holds(LocationRobot_1, s_g, e_g, Going(X, Y))$

Parameter Constraints: $s_g + travelTime(X, Y) = e_g$

Disjunction:

Configuration Rule:

Configuration Interval: *met-by* $Holds(Loc, s_{a1}, e_{a1}, At(S))$,
 $S = X, Loc = LocationRobot_1$

Configuration Interval: *meets* $Holds(Loc, s_{a2}, e_{a2}, At(D))$,
 $D = Y, Loc = LocationRobot_1$

Configuration Interval: *contains* $Holds(Arm_1, s_p, e_p, TuckedIn)$,
 $S = X, Loc = LocationRobot_1$

The above example illustrates the representation of different plan segments and the relations between them. The main interval is described in the "Head" field. The "Parameter Constraints" capture the constraints that are local to the main interval. The "Disjunction" field identifies a set of "Configuration Rules", one or more of which have to be true in the final plan. Each Configuration rule is a conjunction of conditions. The relations can be depicted as a network as shown in Figure 10.31.

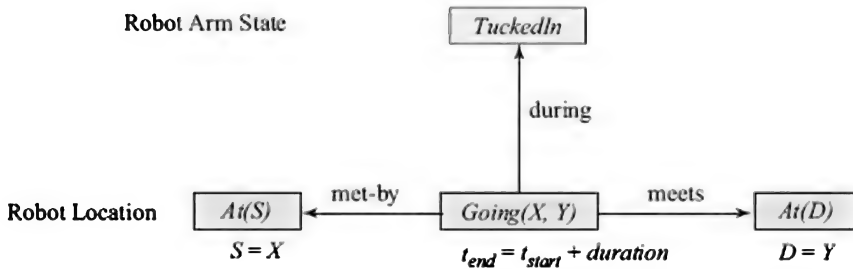


FIGURE 10.31 The Compatibility of the *Going(X, Y)* interval. It must be *met-by* an *At(start)* interval, *meet* an *At(Destination)* interval, and must be *during* the *TuckedIn* interval for its arm.

The relations between intervals can be seen as constraints between the nodes of a network. The corresponding CSP is a dynamic CSP (see Chapter 9) in which new variables (or nodes) are added in the processing of solving the CSP. The new nodes in the context of planning are related intervals that a planner adds to the network (plan), starting with an initial, underspecified network representing the planning problem.

In the above example, there is only one *Configuration Rule*, but one can imagine situations when the Head interval can be supported by different sets of intervals. In the robot example, the *Going* action could be met-by a *Turning* action, wherein the previous "state" of the robot was that it was turning before it started moving. Examples from other domains are as follow. A baby might be *happy* (during an interval) as long as it is being picked up and carried about. It might also be *happy*, if alternately, it is allowed to splash around in water. A trekker in the hills may feel comfortable during the evening as long as she is in the state of wearing warm clothes, or alternately if the group is sitting around a small bonfire. A planner that needs to achieve the state of being comfortable will have to achieve either of the intervals. On a grander scale, a coalition government of a country may be in power (over an interval) either if it is supported by party A, or if it can entice a segment of party B to break away and support it.

We have already observed that there is no distinction between the manner in which an action or a state is treated by the planner²⁶. Another interesting feature of the representation is that the problem and the solution are also described by the same representation. Both are constraint networks. They differ in the fact that the solution is a network that has no unsupported nodes. This is reminiscent of the Plan Space Planning (PSP) approach we studied in Chapter 7. The problem statement in PSP is an initial empty plan Π_0 made up of two actions that we will call A_0 and A_∞ . There is a similarity in the planning process as well, which is driven by the resolution of flaws in the current plan. The input to the planning process need not be an empty plan, but could be an underspecified plan in which the planner has to remove some flaws.

The operations for modifying plans that can be used for searching valid plans are of two types. *Restrictions* add new decisions to the plan or

reduce the number of options remaining for existing decisions. If a *compatibility* rule requires an interval then one can (a) insert a new interval between two intervals of an attribute, along with implied ordering constraints, or (b) add constraints to an existing interval to satisfy the compatibility. In addition, one may insert an unsequenced interval on an attribute, or restrict the values that a variable can take. The inverse of these operations are *relaxations*.

A major advantage of this uniform representation of actions, states, plans and goals is that plan repair becomes more feasible. Combined with the fact that the planner is required to produce a flexible plan instead of a rigid one, the prospect of executing the plan successfully in the real world, making small corrections where needed, become much more feasible. One introduces the notion of a *sufficient* plan to characterize the completeness requirements for the planning algorithm. The planning algorithm needs to build a plan only for a given planning horizon, for example, and may leave some variables to be instantiated at run time by the Executive.

The interested reader is referred to the autonomy papers referred in this section for a detailed description of temporal planning with intervals using constraints, and the interplay between the planner and the executive in dynamic environments.

10.8 Discussion

Research in planning received an impetus with the development of methods like *Graphplan* in the middle of 1990s. Given that even the simplest of planning domains are in PSPACE, the simple search algorithms were never going to go far. One advance that was made with the advent of *Graphplan* was a two stage process. In the first stage, the search space is delimited and some kind of a reachability analysis done on the given problem. Once a solution looks to be feasible, the search space operates on a delimited space. The second interesting development that took place was the defining of domain independent heuristics to guide search methods. The heuristics estimate the distance of a partial solution from a complete one by exploring a relaxed version of the original problem.

The advancements in planning algorithms, and also the increasing computing power becoming available, led to the exploration of richer domains for planning. Starting with metric resources, researchers have explored conditional effects, contingent and conformant planning, and moved on to planning with durative actions and trajectory constraints. The next advancement of soft goals and soft trajectory constraints raises the difficulty level of the problem. Now the definition of a valid plan itself becomes nonrigid. While one may want to achieve all the goals that have been specified, one would be willing to accept plans that do not satisfy some goals. The fact that different soft goals may have differing amounts

of penalties associated with them makes the planning problem an optimization problem of greater complexity.

At the same time, researchers started employing artificial intelligence techniques to sophisticated problems. The most spectacular applications have been in space, but interest in autonomous robots, whether in the domestic environments or in the depths of the oceans, offers considerable motivation for applying planning techniques in the real world. These applications demand that planning algorithms be embedded in larger systems which can sense and react in a real world. Such applications require a greater amount of integration of different problem solving techniques.

They also demand more robust and efficient knowledge representation and reasoning, which will be our focus in the next few chapters.



Exercises

1. Given the following initial state $\{On(A, C), On(B, D), OnT(C), OnT(D), Clear(A), Clear(B), ArmE\}$ and the goal $\{OnT(A), OnT(B)\}$, simulate the *Graphplan* algorithm on paper and generate a plan.
2. Define the algorithms *MutexA* and *MutexP* to determine whether two actions in a layer or two propositions in a layer of the planning graph are respectively mutex.
3. Find all the subgoal sets at the level $k-1$ for the planning graph in Figure 10.8.
4. Write the procedure *RegressGoalSet*($G, i, PlanGraph$) employed in the *ExtractPlan* procedure of Figure 10.10. Your procedure should return the set of all subgoal sets that can be regressed to from the given goal set G .
5. Rewrite the *ExtractPlan* procedure of Figure 10.10, removing the nondeterministic *CHOOSE* action and employing a backtracking like procedure (see Chapter 9).
6. Define the function *SizeSubgoalSets* used in the algorithm *Graphplan* in Figure 10.7. The function should get the value by inspecting the memoized memory *mem* maintained by *ExtractPlan*.
7. Given the set of predicates $(:predicates (clear ?b) (on-table ?b) (empty ?h) (holding ?h ?b) (on ?b1 ?b2) (hand ?h) (block ?b))$ in untyped *pddl*, define the actions *Stack*, *Unstack*, *Pickup* and *Putdown* of the blocks world domain. Express the following problem²⁷ (with one robot hand h) in the language.

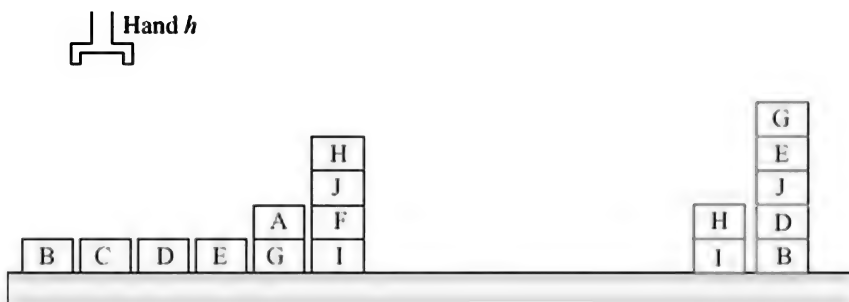


FIGURE 10.32

Download *Graphplan* or *STAN* and run the above problem. Verify that the plan is made up of 16 stages of one action each— $\{unstack(h, H, J)\}, \{stack(h, H, E)\}, \{pickup(h, D)\}, \{stack(h, D, B)\}, \{unstack(h, J, F)\}, \{stack(h, J, D)\}, \{unstack(h, F, I)\}, \{stack(h, F, C)\}, \{unstack(h, H, E)\}, \{stack(h, H, I)\}, \{pickup(h, E)\}, \{stack(h, E, J)\}, \{unstack(h, A, G)\}, \{stack(h, A, F)\}, \{pickup(h, G)\}, \{stack(h, G, E)\}$. Is this a unique plan? Also verify that the goals first appear nonmutex after layer 8, and that the fixed point is reached (graph levels off) at layer 10.

8. In the above problem, replace the single hand h with two hands l and r . Run the program on the new problem.
9. Run the program on the problem suggested by Blum and Furst (1995).

Given = $\{(on\ A\ B), (on\ B\ C), (on-table\ C)\}$
 Goal = $\{(on\ A\ B), (on\ B\ C), (on\ C\ A)\}$
 and observe the program's behaviour.
10. Run the program on Sussman's anomaly,

Given = $\{(on\ C\ A), (on-table\ A), (on-table\ B)\}$
 Goal = $\{(on\ A\ B), (on\ B\ C), (on-table\ C)\}$
 What is the solution found? Is it found before the fixed point or after?
11. Consider the following planning problem.

Domain predicates: oXY, cX, AE, hX -short form
 $on(X, Y), clear(X), AE, holding(X)$

Operators: uXY, sXY -short form

$(unstack(X, Y))$

(preconditions : $on(X, Y), clear(X), AE$)

(effects : $holding(X), clear(Y), \sim AE, \sim on(X, Y)$)

$(stack(X, Y))$

(preconditions : $holding(X), clear(Y)$)

(effects : $on(X, Y), AE, \sim holding(X), \sim clear(Y)$)

Start S_0 : $on(A, B), on(C, D), AE, clear(A), clear(B), clear(E)$

Goal g : $on(A, D)$

Construct a planning graph of two action layers (P_0, A_1, P_1, A_2, P_2)

12. Write the logical bit-level operation that algorithm STAN would do to decide the applicability of an action in a given layer. (**Hint:** What would it mean to check whether an action is mutex with itself?)
13. Given the planning graph of Figure 10.4 and the goals $\{On(A, C), OnT(C)\}$, trace the plan extraction process when the problem is posed as a dynamic CSP. The domains of the variables and the constraints between them may be defined as and when they become active.
14. What is a *relaxed planning problem*? Illustrate with the blocks world domain. Where is it used?
15. Pose the following problem as a planning problem. Find a contingent plan for the problem. *"You are sitting blindfolded and in front of you there is a square horizontal board that can be rotated about the vertical axis passing through the centre. For simplicity, we assume that rotations are in multiples of 90° . Placed on the four corners are four identical objects that can be in one of two states, up or down. You can reach out to any two corners with your hands and can sense whether the two are on a side or a diagonal, and you can sense the state of each object on the two corners. You play a game in which the following moves happen alternately.*
 - An adversary rotates the board by some amount.
 - You are allowed to sense any two objects, and change the state of both, one or none of them. *The goal is to bring all four objects in the same state (up or down). If you succeed, a judge rings a bell."*
16. Find a conformant plan²⁸ for the above problem. What can you say about the lengths of the two plans, contingent and conformant?
17. Extend the planning graphs of Figure 10.12 for a domain with two toilets t_1 and t_2 , which are not clogged in the given state. Show that a conformant solution for the bomb diffusing problem can be found by CGP.
18. Extend the matching diagram of Figure 10.13 to include connections for the actions $Stack(A, C)$ and $PtDn(A)$. Draw the domains for the

new variables as and when needed.

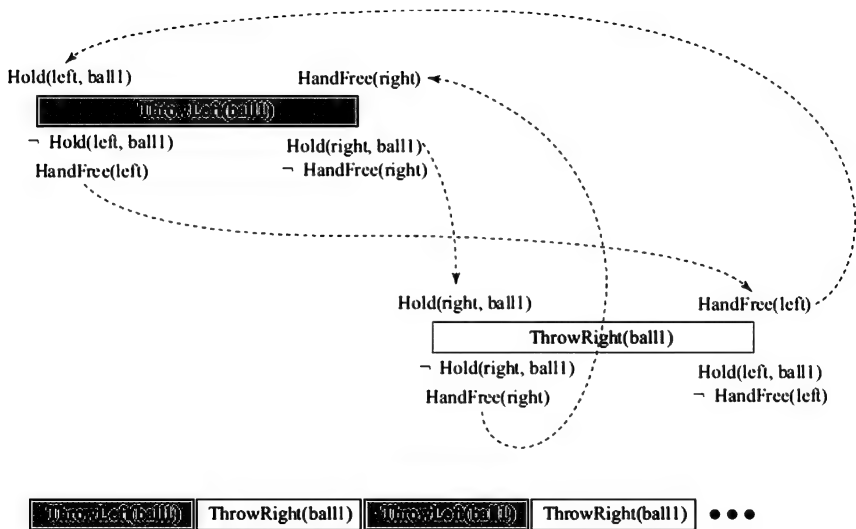


FIGURE 10.33 The two actions *ThrowLeft(ball1)* and *ThrowRight(ball2)* and the relations between them. Shown below is a sequence of actions representing a ball being thrown from one hand to the other alternately.

19. In the MPT representation used in *Fast Downward*, pose the blocks world problem such that the variable *Holding(arm)* is a derived variable, which can take values from $\{A, B, C, \text{nil}\}$. Draw the domain transition graph for this variable when there are three blocks *A*, *B* and *C*, in a planning problem.
20. Draw the causal graph for the planning domain depicted in Figure 10.18.
21. Model the operator for driving a car from a point *A* to a point *B* as a durative action with metric fluents. Assume that distance from *A* to *B* is given, the (average) speed of the car, and the rate of guzzling fuel is known.
22. *The spy-swap problem.* Two unnamed countries have a desire to swap two spies named Black and White. Black is being held at location *M*, while White is a prisoner at location *N*. The swap is to take place on a bridge *B* over a river that divides the areas controlled by the two countries. *B* is reachable from *M* and *N* over a road network, available as a graph weighted with distances. It takes five minutes to walk across the bridge. Being suspicious of each other, the countries would like Black and White to walk across at the same time. Pose the above problem as a temporal planning problem.
23. Given instance of the two durative actions *ThrowLeft(x)* and *ThrowRight(x)* (from (Coles et al., 2009) shown in the Figure 10.33, show how the plan of a robot juggling three balls would look like.
24. Planning operators in the PDDL family of languages are expressed

using preconditions and effects. Take an example operator and show how it can be represented in the *Compatibility* formalism defined in *CAIP*.

- ¹ They can, however, be encoded as new positive propositions. For example, the fluent *clear(x)* in the blocks world domain is equivalent to the condition $\bigvee \exists y (on(y, x))$. The fact that actions explicitly add and delete this fluent makes it unnecessary to verify the equivalent form logically.
- ² We have adopted shorter names for some predicates to facilitate drawing planning graphs. Here, *Hold(A)* is a shortened form of *Holding(A)*.
- ³ Occam's razor. See http://en.wikipedia.org/wiki/Occam%27s_razor
- ⁴ Comparing the sizes is sufficient. The two sets will be equal as well, but checking for set equality is more expensive.
- ⁵ A mandatory check in most Indian cities.
- ⁶ As of writing, this PDDL does not cater to nondeterminism in effects.
- ⁷ Which is equivalent to $\neg[Hold_1(C) = PkUp(C) \wedge Hold_1(A) = UnSt(A, B)]$ or $\neg(Hold_1(C) = PkUp(C)) \vee \neg(Hold_1(A) = UnSt(A, B))$
- ⁸ Numbered in the same manner as the planning graph.
- ⁹ As opposed to classical predicates in FOL that are either true or false, once and for all.
- ¹⁰ We will stick to our planning graph based numbering. The numbering adopted by Kautz and Selman in the original papers was different.
- ¹¹ *ArmEmpty*, the proposition used in STRIPS, can be thought of being equivalent to *Holding(nil)*.
- ¹² Again, we stick to *Graphplan* styled numbering of layers.
- ¹³ We can use the sum of the costs of actions, if actions have costs associated with actions.
- ¹⁴ The weighted *A** algorithm uses the function $f(n) = g(n) + w \cdot h(n)$ to pick nodes for expansion.
- ¹⁵ We continue to use *A* for the set of actions. The papers use the term *O*, for operators, also commonly used elsewhere. The paper uses *A* for axioms which we have replaced with *D*, for derived axioms.
- ¹⁶ And hence the name 'Downward'.
- ¹⁷ There are people who will vouch for this.
- ¹⁸ The description is only illustrative. Not everyone uses ghee. And *chapati* making is a more complex action, better modelled as a hierarchical compound action.
- ¹⁹ "We impose the further constraint that no logical condition can both be required to hold and be asserted at the same instant" – (Fox and Long, 2003). Hence, they cannot finish together.
- ²⁰ The standard example used in literature requires a light-match to make light available at the end of a mend-fuse action that is longer in duration.

- 21 The literature on *Sapa* uses the term delayed effects.
- 22 In July 1994, the comet broke up into pieces and crashed into Jupiter.
See <http://www2.jpl.nasa.gov/sl9/>
- 23 See also <http://ti.arc.nasa.gov/tech/asr/planning-and-scheduling/remote-agent/>
- 24 "A teleo-reactive (T-R) program is a mid-level agent control program that robustly directs an agent toward a goal in a manner that continuously takes into account the agent's changing perceptions of a dynamic environment." — Nils Nilsson
<http://ai.stanford.edu/users/nilsson/trweb/tr.html>
- 25 The Event Calculus is a framework for reasoning about events and change, and has predicates like *Happens* and *Initiates*. In CAIP, only one such meta-level predicate *Holds* is used to assert that an action happens or a state persists over a time interval.
- 26 The Executive though, as expected, treats them differently. Actions are what it needs to activate in the agent and states are observations it gets.
- 27 Thanks to I. Murugeswari for the example where a plan is found *after* the fixed point.
- 28 First demonstrated to me many years ago by Vinay Desai, a fellow student and a bridge player at IIT Bombay.

Knowledge Based Reasoning

Chapter 11

The goal of artificial intelligence is building systems that will come up with a good answer for any question from its domain of competence. AI systems are different from specific application programs in the *scope* of questions they can address. A program to multiply two matrices can only answer the question: “What is the product of these two matrices?”, and that too only if the input is presented to it in a carefully tailored form. It does have knowledge about multiplying matrices, but that knowledge is embedded in the algorithm. The knowledge for problem solving is encoded in a procedural form. It does not know what matrices are.¹

The traditional approach to software development is to implement more and more such programs for each task that is to be done. Each such program encapsulates a connection between its input and its output. Each is confined to that particular connection. This relation between input and output may be complex, allowing many choices to the user. Consider, for example, a spreadsheet, a word processor, or a media player. Each of these programs may give a multitude of *fixed* options to a user. A spreadsheet, for example, allows a user to arrange a lot of data into grids and define functional relationships between them. Word processing software allows the user to organize and format words and pictures, taking upon itself a lot of mundane chores like alignment, styles, and spelling checks. A media player may enable a user to create, organize and access movies and music in various formats. In each of these cases, the connection between input and output is fixed. And each program is disconnected; oblivious of the others, using its own internal representation, even for data that may be common to different programs. Furthermore, it is the user who has to *choose* the appropriate program given a problem to solve, and tailor the data to execute it successfully to generate the answers. Sometimes, the answers themselves may have to be interpreted. As we move away from a matrix multiplier to a system like *Matlab* or *Scilab* or R^2 , the user gets to establish more and more connections between input and output using the same piece of software.

The endeavour in creating an artificial intelligence is to extend this diversity to a wider set, perhaps including connections that have not been hard coded into the algorithms. In addition to expanding the set of possible connections between input and output, the goal is to also make these connections transparently, that is, without the need to user intervention. Figure 11.1 illustrates the difference between traditional

programming and the AI approach. In traditional programming, a programmer implements the solution steps for problems in a given domain into a program. The program thus embodies the problem solving knowledge of the programmer. In AI programs, on the other hand, the programmer implements a problem solving strategy in a more domain independent form. The resulting AI program takes a domain description and a given problem and produces the solution steps, based on the strategy it embodies. The reader would recall how planning problems and the domain description are both described using a planning domain description language (see Chapter 7). Both the problem solving strategy and the domain description, call upon knowledge representation approaches.

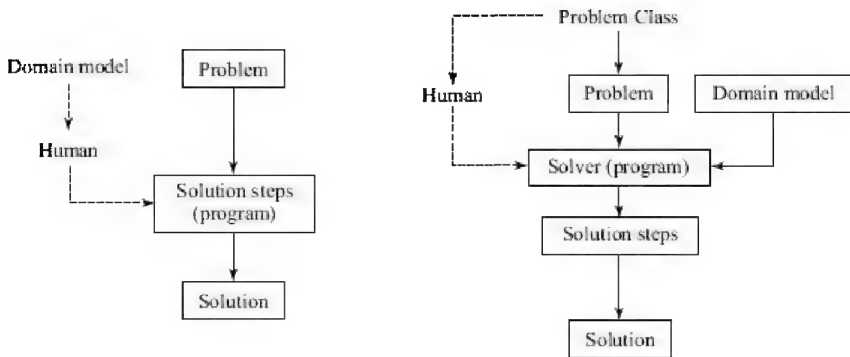


FIGURE 11.1 In the traditional programming approach shown on the left, a human being programs the solution steps for a specific domain. In the AI approach, the human programs a solver for a class of problems, and given a domain model and a problem instance from the domain, the solver produces the solution steps that lead to the solution.

One distinguishing feature of artificial intelligence programs is that connections between input and output may be found at runtime, and not encoded by the programmer. This is useful in domains where the number of individual connections may be too many to enumerate and too diverse to generalize. Consider for example, path finding on a city map. Given any reasonable sized city, the number of origin-destination combinations may be too numerous to enumerate. And unless the city is a regular grid like Manhattan (and the coordinates of the two points are known), it may not be feasible to devise a greedy path finding strategy.

The option we have explored so far in problems like these is to *search* for the solution (path). Search is but one of the means to finding a solution. We need to also look at how experience often leads us down an oft trodden path. Travellers often rely on guidebooks and travelogues and exploit the experience of those who traveled before them. For a problem solving agent, this experience has to be captured into some form of (knowledge) representation. Let us look at another example. Consider the problem of finding the meaning of a set of uttered words. Again, the permutations of words are too numerous to enumerate. Parsers embody

some generalized knowledge of the relations between the different constituents of a sentence. The meaning itself derives from the meaning of constituent elements of a sentence. The problem in natural language understanding is further compounded by the fact that the meaning of a sentence may depend upon context (like a neighbouring sentence). For example, "Suresh was in a hurry to catch his flight. He ran towards the terminal." Reading a piece of literature may often require the reader to keep many segments in context, and in serial fiction like Sherlock Holmes, or the more recent Harry Potter, the reader may have to make sense of a piece of text based on what was written in an earlier story.

Another scenario where artificial intelligence approaches are called upon is when the amount of computation required for establishing the connection between input and output is beyond acceptable limits. Consider the example of a chess game. Given a board position as input and a question about the outcome of the game, one needs only choose between three possibilities; a win or a draw or a loss for white. Alas, as we have seen in the game playing chapter, while this can be computed in principle, it is not feasible in practice. Another example of a hard problem we have seen is the travelling salesman problem. Faced with such problems, our approach so far has been the introduction of knowledge in the form of heuristic functions to cut down the search space. Humans on the other hand rarely do systematic search, and tend to rely more on knowledge and experience. Of course, they have the benefit of a lifetime of learning to fall back upon, and the experience of others in the form of advice and books. In the following chapters, we will explore how domain knowledge can be represented in a way appropriate for the reasoning tasks, and also how problem solving experience itself can be represented, accrued, and exploited.

Humans resort to knowledge and experience extensively while solving problems. Human memory is the seat of this knowledge, and we refer to this problem solving approach as memory based reasoning. How does all this knowledge find its way into the memory? The most straightforward way is by simply storing experiences. Experiences may be stored directly, or through a process of generalization and modularization and are expressed in the form of rules of thumb. Figure 11.2 depicts the architecture of a knowledge based problem solving agent, that predominantly uses knowledge to solve problems, but if the need arises, invokes a first principles search based approach to solve problems it cannot solve directly using the stored knowledge. Experience accrues as the agent solves more and more problems.

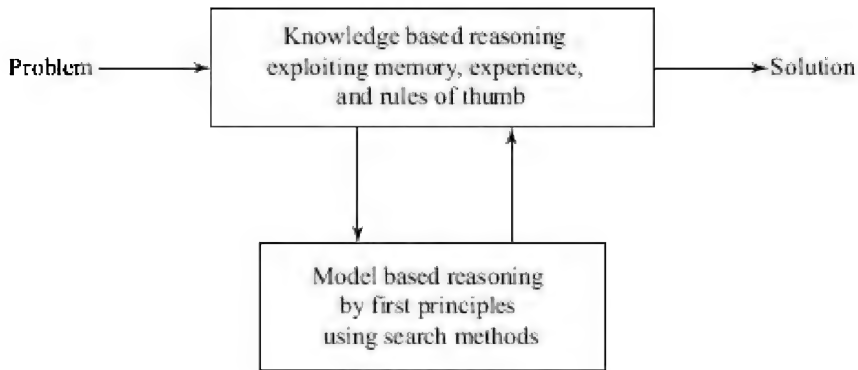


FIGURE 11.2 Humans primarily use knowledge to solve a problem. Often “we know” the solution for a familiar problem. When we do not then a first principles approach based on search is the recourse. The first principles solutions can be stored in the memory for later reuse.

The agent’s own experience is not the only source of knowledge. The human species has thrived and become dominant because we have learnt to share knowledge, and have invented languages to do so. Traditionally, storytelling has been a form of knowledge transfer in all societies. Similes, metaphors and idioms are devices to capture nuggets of distilled knowledge. One can benefit from the accumulated knowledge of an author far removed in time and space simply by reading her book. Languages like English, Urdu and Kiswahili have been the languages for knowledge representation for us humans. But computers are not yet completely natural language enabled. They need crisp, unambiguous and precise formalisms for representing and manipulating knowledge. In the following chapters, we will explore various issues in representation and reasoning with knowledge, the role of memory and language, the formation of concepts, approaches to deal with uncertain knowledge and machine learning strategies to acquire knowledge.

11.1 Agents

The need for knowledge representation will also arise if we want to devise autonomous systems. Such systems, known as *agents* (Woolridge, 2000), would have the following properties. They would exist for extended periods of time (are persistent), would have broad goals from which they would devise immediate goals (are goal oriented), would sense the surroundings, invoke goals, and act (are situated and pro-active), and would communicate with their owners and other agents (have social ability).

Such programs would be running all the time, deciding themselves upon their course of action, in contrast to a user invoking a program with a specific input and required output. For an agent to act out its decisions, it must be embodied in some environment. It is obviously easier to

implement agents in cyberspace³, circumventing the difficulties of perception in a physical world. We are all aware of viruses, worms and other forms of malware let loose on the internet.

However, there have been efforts to build physical agents as well. The most famous was the Remote Agent architecture (Muscettola et al., 1998) in NASA's Deep Space One spacecraft. The need for autonomy here is acute because it takes a long time for information to travel over astronomical distances, and by the time a human on Earth senses something and reacts, and the command reaches the spacecraft, it may be too late. Autonomy is also useful on satellites orbiting the planet keeping a watch on storms, floods, fires and other phenomena. More recently, there have been efforts to build teams of robotic agents for autonomous activities like search and rescue operations (see for example (Alboul et al., 2010), (Marjovi et al., 2010) and (Meyer et al., 2011)).

At the very minimum, such a program must have a set of goals or tasks to perform and be able to sense its environment. Ants, for example, are a well studied example of such simple agents. But we are in quest of a higher level of intelligence. This would require that the agent is *aware* of its resources and abilities, and be able to make informed judgments. Further, it should learn from its experience⁴. Learning from experience is a slow process, even for the quickest on the uptake. It may not be enough to rely on one's own experience. For an agent to emulate human level of performance, it will need to benefit from shared knowledge accrued over generations in societies.

11.1.1 Belief, Desire and Intentions

One of the popular approaches to devising agent based systems is to build them on the *Belief-Desire-Intention* (BDI), first expounded by Michael Bratman (1987; 1990). According to this model, rational agents will need to have three kinds of information to act in a rational manner.

The *beliefs* of an agent correspond to what the agent knows about the world. A rational agent would have a model of the world in its head, which would facilitate reasoning required to produce rational actions. Sophisticated agents would have a representation of themselves in the model of the world they hold. Rationality is tied up with goals of the agent, and in that sense is concerned with finding the actions or decisions that are optimal for achieving the goals. The goals are themselves described in terms of desires and intentions.

An agent's *desires* represent the things that the agent would want to happen or be in world. The set of desires may not be consistent or even feasible⁵. An agent may well have a desire that is not feasible. It may want to know whether White wins in the game of chess, or it may want to see a unicorn, or hop one legged up to Mount Everest if it is physically embodied. It may have a set of desires that may be individually feasible, but be inconsistent with the others. An agent may want to play a game of

bridge, clean up the garage, have a long siesta, finish writing a paper, or go for a walk in the park, all on a Sunday afternoon.

The *intentions* of an agent refer to the subset of desires that the agent decides to act upon. Thus, one aspect of rationality is in choosing the intentions. This may involve only choosing goals that are feasible. This may also involve choosing short term goals that are consistent with long-term goals. For example, one may choose the goal of finishing a paper, as opposed to playing a game of bridge, if one is studying to earn a degree. Rational behaviour may also mean optimizing the number of desires one can achieve by choosing intentions and actions (planning) appropriately. For example, one may go to the park and work on the paper writing there.

Intentions of an agent can be referred to as plans as well, though not in the sense of a plan being a structured representation of actions as in Chapters 7 and 10. Rather, the notion of a plan is that of a mental state, in the sense that “a plan to roast lamb” is the intent to roast lamb, as opposed to “a plan for roasting lamb”, which may be a recipe for the goal (Bratman, 1990). That is, the agent has a plan or intention to achieve the goal of having roasted lamb. If for a given goal G , an agent has a plan $\Pi = (a_1, a_2, \dots, a_n)$ for achieving the goal then the agent also has the *belief* that the actions a_1, a_2, \dots, a_n that make up the plan will eventually lead to the achievement of G (Pollack, 1990).

Matters become complex when the world is dynamic. Time and change complicate the rational process. This may happen because change is a part of the world. Like the day and night cycles we are used to, or the changing weather, seasons, our growing bodies, and many other phenomena in the natural world. The world is changing also because of the actions of other agents.

This changing world has myriad ramifications. Firstly, the plans we generate may not remain correct because something has changed in the period in which we planned and commenced execution. You may have set out for a bicycle ride, only to find a flat tyre. You may have to modify the plan or devise a new one. Secondly, the intentions (goals) we chose earlier may no longer be relevant because something has changed. For example, you had the intention of buying a loaf of bread, but on reaching the bakery you spot a piece of pizza that you decide is a better snack. Thus, new desires may arise as things change in the world. Thirdly, an agent needs to keep revising its beliefs in a changing world, and how often it does so may critically depend upon the intentions it is pursuing. If you are flying a kite, you may need to keep a close watch on the wind and other kites. If you are playing a game of tennis then you need to keep an eye on the ball at all times. On the other hand, if you are playing a game of bridge, you only need to look out for the next card played, unless it's the opponent's expression after you have put her on test by leading the jack. If you are baking a cake then you could in fact turn away and have a siesta while it is being done. The point is that a rational agent in a dynamic world has to keep doing the following.

- Update its beliefs or its model of the world as it changes.
- Update its desires as new ones crop up and existing ones are achieved or discarded.
- Update its intentions to choose the ones to act upon currently.
- Update its plans as all of the above change.
- Choose and execute an action (from the plan).

Also, all of the above are inextricably cross linked with each other.

Current research on agents focuses less on the knowledge representation aspect, and more on the reasoning processes described above. The research described in agents' literature is more concerned with maintaining a consistent set of beliefs, and optimizing upon the selection and consummation of intentions in a framework, where both computation and memory are resource bounded. The research on knowledge representation has been done elsewhere independently. We will look at the various aspects of knowledge representation briefly below, and in more detail in individual chapters. Let us first try and reconnoitre the different approaches to knowledge representation.

11.2 Facets of Knowledge

Chess playing was one of the earliest pursuits in artificial intelligence, and now there exist many programs that can play better than most human players. Yet, we are not willing to call such a program intelligent. This is probably because we see it as a program that somehow searches through combinations efficiently and nothing more than that. It only knows how to choose a move, given a board position.

Let us try and imagine what would make it seem worthy of being called intelligent. If the program were more *like a creature* (Grand, 2001) then humans are more likely to accept⁶ it as an *intelligent creature*. If such a creature had a chess playing ability, had some kind of a sense of identity, and was in some sense aware that it was playing chess, if it could converse in natural language, if it could comment upon your moves, if it had a memory of past interactions, if it was connected to other things besides chess ("the weather is too good to play inside"), if it had internal goals tied up with emotions and moods then such a program would surely command more respect. Such a program *would know* a lot. How can we build such an intelligent and articulate chess playing agent?

An intelligent program must have a considerable amount of knowledge and it must have the faculty of some kind of language, a means for knowledge exchange. What kind of knowledge would the chess-playing agent need? It must know that chess is a game that "it" is playing. It must know that there are other things than chess. How does it know that "it", amongst other things, is there "playing chess"? It must have a model of the world and must know facts in the world. It must know about the processes that go on in the world and the changes they ensue.

To know a world is to have a representation of the concepts involved and the relations between them; for example, “this is an apple” and “apples are sweet” and “apples are fruit” and “fruits have seeds” and therefore “apples have seeds”, and so on. Such knowledge is traditionally referred to as an *ontology*⁷. We can also call such knowledge as *semantic knowledge* or knowledge about the meaning of terms. Ontologies are categorizations of things, defining them relative to each other, separating categories into classes based on the differences between them, and clustering them together under common classes based on similarities and shared properties. Some of the recent work in ontologies has been motivated by the designs of the next generation *semantic web* and representation using *description logics*. We will look at these concepts in more detail in subsequent chapters.

The other kind of knowledge that we have is *episodic knowledge* or knowledge about things that happen, or events. Such knowledge is rooted in experience and memory, and relates different events together. For example, one might remember that eating an apple makes one feel better when one is hungry, and therefore eating an apple is a solution if the goal is to be not hungry, or that when you go and sit in a restaurant, it is normal and expected for someone to approach you to ask what you would like to eat. Such knowledge allows us to go about the world without explicitly going into a search based quest to determine consequences of incoming information or our proposed actions.

The fundamental question from the AI perspective remains though. How do we *represent* an apple or a carrot? When we think of a carrot, the first thing that often comes up is a visual image in our minds. Visual images are some things *conjured* up by our minds, rather than images received and represented in our brains. We are still far away from figuring out reasoning at the visual level. The image of a carrot in turn usually triggers other *memories*, for example the image of Bugs Bunny⁸ leaning against a fence, carrot in hand. Our *representation* of the concept of a carrot would depend upon what the *purpose* is; if we want to draw it, it has shape and colour; if we want to put it in a salad it has grateable structure; if we want to talk of its food qualities, it is composed of different vitamins, minerals, etc; if we want to talk of cooking, it is associated with recipes. Besides, we can even think of it simply as a physical object that can be thrown and caught, as a living thing that can grow, as a container of refreshing juice. So one might say that a carrot is what a carrot does. It is a snack, it is a vegetable, it is an object, and it is a living thing. You could use it for food, or a paperweight, a subject for a painting, or even a weapon to throw at someone. Each such aspect would relate it to other things, by shared properties. We will explore *frame based* representations of concepts in Chapter 14.

11.2.1 Memory

The knowledge that we have resides in our memory. Memory is what we remember or recall. Memory has long been recognized as the seat of knowledge and has been the subject of study for ages. Since the Greek times, human beings have sought techniques for improving memory (Yates, 1966). Since then scholars from many disciplines, psychology, biology, medicine and most recently, the cognitive neuroscience have attempted to study the structure of the brain and the functions of memory. And we, the artificial intelligence community, are waiting to reap the fruits of such research to contribute to the design of machines with minds.

The field of cognitive psychology views the human brain as an information processing device which equips the human with knowledge and reasoning capacity to solve problems. The prevalent view is that the brain is structured and made up of different modules, each with its own domain and processes of reasoning (ten Berge and van Hezewijk, 1999). This view has been supported by evolutionary biology as well. "*The mind is a squadron of simpletons. It is not unified, it is not rational, it is not well designed or designed at all. It just happened, an accumulation of innovations of the organisms that lived before us*" (Ornstein, 1991). This view has found expression in artificial intelligence too, with Marvin Minsky (1998) proposing a *society of mind*.

Towards the end of the last century, it became evident that there was a distinction between two kinds of memory, one that is accessible to conscious recollection, and another that is not (Squire, 1987). The former is referred to as declarative in nature, and is primarily the subject of interest from the symbolic knowledge representation point of view in artificial intelligence. The second, originally called procedural, is concerned with all kinds of learned reflex behaviour. The declarative memory is concerned with knowing facts and figures, while the procedural memory is concerned with knowing how to do things. Figure 11.3 depicts the basic taxonomy of memory described by Squire (adapted from (Rose, 1998)).

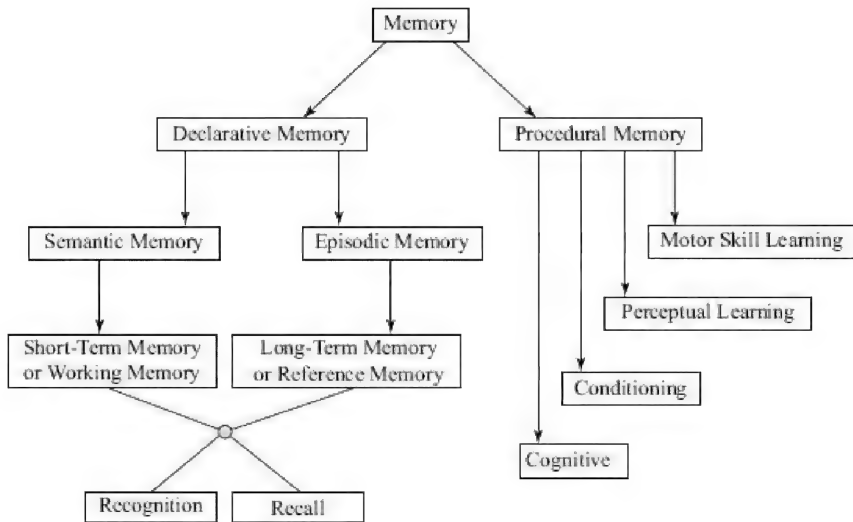


FIGURE 11.3 A neuroscientist's view of the taxonomy of memory. figure adapted from (Rose, 1998).

The procedural memory in the figure is long term in nature. It is the kind of knowledge that is involved in motor skills or perception that is acquired by a process of repeated learning that results in modification of some performance system. It is a form of learning that is invoked by the reactivation of certain part of the brain. Examples of such learned behaviours are riding a bicycle, swimming or even singing. These skills are hard to express verbally. Such learned behaviour would be part of a control system of a robot. *Conditioning* is the process of *acquiring* the kind of information that the brain sends to the body for an automatic response. *Priming* is the nondeclarative memory function that improves the brain's ability to detect, identify, or respond to a stimulus that it has processed recently (Squire, 2007).

Declarative memory, on the other hand, involves explicit representation, which allows remembered material to be retrieved and compared. It is the memory traditionally associated with the notion of knowledge, and it is what is lost in amnesia. In the figure there is a distinction between semantic memory which is concerned with facts, and is associated with short-term memory, and episodic memory which is concerned with remembering events, which goes into long-term memory. The episodic memory is the one that is often lost during Alzheimer's disease. This distinction is subtly different from the one proposed by Newell and Simon, when they identify knowledge in the form of rules as long-term memory and facts in the working memory as the short-term memory (see Section 6.6). One might say that the rules form an episodic memory of how to solve problems.

Declarative knowledge is not conscious till it is retrieved, following some questions or cues. The retrieval process itself is not consciously

accessible. An individual can only become aware of the products of this process. It is also a very selective process. A given cue will lead to the retrieval of only a very small amount of potentially available information. (ten Berge and van Hezewijk, 1999). It has been demonstrated that the process of recognition is much easier than the process of recall. Typical memory games involve *recalling* a sequence of numbers or names, and most of us give up after seven or eight sized collections. On the other hand, we have no difficulty in *recognizing* a face from the thousands we might have seen.

In Chapter 15, we investigate the memory based approach to solving problems in which a key issue is the retrieval of a relevant piece of memory.

It is interesting to note that computer programming also has paradigms reflecting these two kinds of memory or ways of encoding knowledge. In the imperative paradigm, a computer program is a sequence of instructions, with loops possibly built in, to be followed. The program is a procedure for solving a program. In declarative paradigms like logic programming, the program is the knowledge of relations between in a domain. A high level procedure operates on this declarative knowledge to solve problems.

The insights gained into memory owe much to the study of patients who have suffered accidents, and brain lesions caused by strokes (Rose, 1998). One such famous patient known only by his initials, H.M., was operated on in 1950 for epilepsy in Montreal. During the operation, the patient's hippocampus and parts of his temporal lobe were removed. Following the operation, H.M. lost the capacity to create new long-term memories. He could only remember events for a few minutes. More recently, Ramachandran (for example in (2010)), describes how a patient called John suffered a stroke as a result of which he lost his ability for visual recognition. He could not recognize his wife's face, even when he could recognize her by her voice. He even had to convince himself that the face he saw in the mirror was his own because "*it winks when I wink and it moves when I do*".

More recently, with new imaging techniques like PET (Positron Emission Tomography), MEG (MagnetoEncephaloGraphy) and CAT (Computer Aided Tomography), scientists have been able to map the different memory functions to brain regions. Figure 11.4 depicts this mapping of different kinds of memory functions, across both types, and how they are supported by specific regions of the brain as described in (Squire, 2004). As observed by Squire, over the last two centuries, the study of memory and the human brain has shifted from the fields of philosophy and psychology to biology.

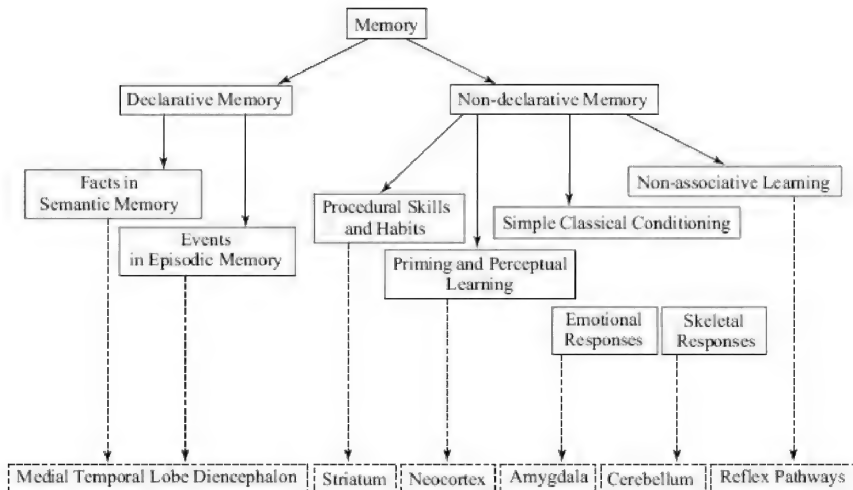


FIGURE 11.4 The mapping of different memory functions to the regions of the brain. Figure adapted from (Squire, 2004).

11.2.2 Declarative Knowledge

As humans, we often do things without conscious perception or conscious planning. This would be ascribed to procedural knowledge, which is a form of control knowledge a system learns. However, in classical or symbolic artificial intelligence, we are more concerned with the kind of knowledge that can be represented explicitly⁹. Our focus in this book is on declarative knowledge.

The simplest form of semantic knowledge is relations between entities of a domain. This is a well-studied form of knowledge representation with formal logic as the device both for representation and reasoning. The development of logic was more concerned with fidelity of reasoning; how does one make inferences that are irrefutable? In Chapter 12, we begin by studying the machinery of logical reasoning. However, reasoning is done over some symbolic representation and the symbols of something. Human beings have traditionally used natural language as a medium of representation as well as communication. In Chapter 13, we try and bridge the divide between logic and language and explore how concepts can be represented in logic. Given that our knowledge representation is about the world we are concerned with, and that the world is structured, it is only to be expected that the representation would start to mirror the structure in the world. We introduce abstractions and generalizations. We explore such structured representations, in which fact do not exist merely as individual statements but are woven together in schemas or schematas in Chapter 14.

Episodic knowledge accrues from our accumulated experience, but is not confined to that. It may evolve into other forms of knowledge; by

processes we now call *data mining* and *knowledge discovery*. The semantic notion that “apples are sweet” may in fact be rooted in the episodic memory of having eaten an apple in the past and experienced the sweet taste. We constantly endeavour to *generalize* from instances of experiences and make compact *rules* and *associations* that are useful for us. For example, traditional wisdom says that it is not a good idea to go for a swim after a sumptuous lunch. Often, knowledge gets encapsulated in such rules, and may even disassociate from the original experience, leading us to wonder about the etymology of phrases like “to bury the hatchet” or “kick the bucket”. An ironsmith may associate a particular colour of the heated metal with a particular action, and if he has deeper knowledge, then with a particular property of the metal. Machine learning has always been a fascination for computer scientists since the day when Samuel’s checkers program learnt to play better and better and eventually beat him. We look at some of the techniques developed in machine learning in Chapter 18.

Over a period of time, we evolve such generalized experiences into *action schemas* or *procedural rules* that we use in our everyday problem solving. For example, a farmer may say that the correct time to sow his crop is after the first rains in the monsoon, or that a red sky foretells a stormy night. Grandma cures for common diseases may dictate that one must “starve a fever and feed a cold”. Over a period of time, such knowledge, known as *heuristic knowledge* accrues, and we begin to form notions of how things work and how to solve problems. Your grandmother may have discovered that tapping her radio set at a particular location was instrumental in getting better reception, and your cousin would have the policy of rebooting her computer system at the slightest hint of slowness. Heuristic knowledge is essentially distilled from experience. It embodies a shortcut between pattern (problem) and action (solution). It may be wrong sometimes, in the sense that it may not always be true to reality. But it often is, and that is why such knowledge has survived in competition with other kinds of knowledge.

As we get more *scientific* and as our knowledge about the world increases, we build detailed *models* of things and processes. Models are representations of reality that help us predict how things will pan out in reality. They allow us to design machines and processes that will bring about what we want to happen in reality, because at some level they simulate reality. One also refers to models as deep knowledge as opposed to shallow heuristic knowledge.

Models can take very different forms. They may involve different forms of mathematical tools, or models may be simplified physical artifacts. Thus, an aircraft designer may take inspiration from small physical models, or may do closer to reality wind tunnels experiments to study the lifting ability of a wing design. She may also construct a finite element model on a computer system, or work with other forms of mathematical formulas with a pencil and paper. Models also allow us to understand the world around us and explain phenomena like the rain, and the movement

of the sun and the stars in our sky.

Models are representations of reality¹⁰ that are acceptable as long as they are useful. For humans, the sun used to go around the earth till Copernicus found a new model that *explained* the motion of heavenly bodies better. The world used to be flat until Columbus embarked upon a journey around the world. Newton's laws of physics are good for everyday world dynamics, but break down at the sub-atomic level. Einstein showed that even our notion of time flowing at a constant steady pace is not really correct¹¹. Models can reflect reality with varying degrees of fidelity. Models can also be at varying levels of detail. The more detailed and accurate they are, the more work (computation) they require to make predictions.

Consider the well-known example of computing the time that an object will take to go from point *A* to point *B*. We could model the movement of its different parts and integrate the results. For example, if the object in question is a boy on a bicycle, we could twist ourselves silly, reasoning about the movement of a point on the rim of the wheel, or the movement of his knee. More sensibly, we tend to model the boy on the bicycle as a point moving in a (hopefully) linear trajectory and apply a simple formula to determine the time it would take him. But this simple model does not tell us how his weight applies varying amounts of pressure on the two tyres, as he rocks forward and backward gaining momentum; and how the bicycle swings from left to right and back with his frantic pedalling. If we required that information, we would need a considerably more complex model, requiring much more computation.

When doing modelling, we have to make a trade-off between the amount of detail in the model that the computation will work with and the accuracy of the predictions, and usually we only choose the level of detail that is sufficient for our requirements. One area where models have been becoming more and more complex is in weather forecasting. We are no longer content with farmers' heuristics; instead, we build increasingly detailed models in powerful computers feeding it data collected from a large number of locations on the earth; thus enabling the metrological departments to forecast more accurately whether the next weekend will be a suitable day to spend outdoors or not.¹²

Can we construct models that will be *perfectly* accurate? The scientist Edward Fredkin postulated that our universe *is in fact a simulation* (Wright, 1989). As a corollary, it follows that while it would be possible to construct such a model and run it, it could only make predictions as fast as (actual) reality. That is, even if we could implement a perfect model, it would make perfect predictions but not before they actually happen, because the model would have to replicate what is happening in "reality".

Human beings, on the other hand, have no time for complex models. We do not use "rocket science" to make predictions in our daily lives. The typical human being senses pangs of hunger and heads towards the refrigerator, or towards the local market to buy some fruit. In fact, very rarely are our day-to-day models mathematical. Instead, they are what

we now call *qualitative models*. We know that as we eat more, our hunger becomes less, eventually leading to a point of satiation. When we pump air into our bicycles, we do not reason with numbers or formulas. We pump till we *feel* that the pressure is enough. We do not monitor the temperature of water in a pot of tea we are making. We predict (know) that on the stove, the water temperature will rise till it comes to a boil. We know that if we are pouring tea into a cup, the quantity in the cup will increase and eventually it could overflow. We do not compute the rate of fluid flow or the volume capacity of the cup. Human beings do a fair amount of *qualitative reasoning*.

We also build qualitative models of the world around us, and our interaction with other people. A significant amount of our reasoning combines factual knowledge combined with qualitative reasoning. We know that as winter approaches, it will get colder¹³. We know that watermelons are available in the summer. We know that people get hungry often in a day, though we do not compute times. We reason about emotions at a qualitative level. We know that children are happier when they get to play. We can predict that if you help someone, they will be happier. We feel better when others smile at us, and most of us generalize from this experience and smile at others. We also have specific knowledge about people, places and things. We know that the beach is crowded on weekends. We know the likes of people close to us. I know my daughter is a Djokovic fan. I know that she and her friend love pasta. I know her friend's dad is a movie buff.

Qualitative reasoning is but one of the approaches we employ to reason in a world with incomplete and uncertain knowledge. The other techniques are default reasoning and probabilistic reasoning. We look at some of these techniques in Chapter 17.

The point is that when we exist in our world and act intelligently, interacting with others and the surroundings, we do so with the aid of lots of knowledge of different kinds.

We have factual knowledge (so popular amongst quiz masters); we make inferences, sometime erroneous, about what we do not directly know; we have knowledge of how to do things; how processes change the world. Over different domains, we have knowledge at different levels of detail and fidelity. We do so with all this knowledge in *one place* commonly shared for all the “applications” in our minds. It will still be a while before our artificial intelligence systems can do the same. But we are on our way. We are exploring the different forms of knowledge representation and schemes to reason with them.

We are exploring the development of layered architectures, where the lower levels will provide meaning and functionality to higher levels. This process has been on since programming languages have moved up the ladder of abstraction. While a human looks at a city map and “sees” a path to his destination, we may build a program for which a search function at a lower level does the same. And then it could carry on with whatever it was doing at the higher level. To the higher level, it should not

matter how the lower level does compute the answer.

Given an input (problem), the output (solution) may be found either by search or from a look-up table possibly embedded in an algorithm. Imagine the game of tic-tac-toe (also known as noughts and crosses). One could play it either way. One player may mentally project moves into the future and make a choice. Another may have a look-up table for each position. A third may have some general heuristics. To an opponent, the distinction may not be obvious. In fact, the way humans play chess is decidedly different from the way computers do, and yet they manage to have an interaction with each other. The interaction is functional in nature. We consult websites like google.com and dictionary.com, and do not care how they arrived at the answers we were looking for.

We can view the vastly diverse kinds of research being done in artificial intelligence as development of building blocks, on top of which more powerful systems will be implemented. It may take a while for each technique to mature, but eventually we hope we will be able to put them altogether to work on a common representation.

And finally, if we have to build computer systems that interact intelligently with human beings, they have to be endowed with the faculties of language. Interest in written language processing, or text processing, has ballooned since the development of the World Wide Web. We look at some of the techniques in Chapter 16.

Meanwhile, each of these approaches is leading to applications that exploit the particular developments. They may not be self aware intelligent systems, but they are serving useful purposes in different applications. While looking at different representations in the following chapters, we will also look at the use they have been put to, but we shall keep in mind that in the future, we would want to integrate them in one knowledge base.

¹ In fact there is no “it” that will know anything at all.

² See <http://www.mathworks.in/products/matlab/> and <http://www.scilab.org/products/scilab/> or <http://www.r-project.org/> respectively.

³ For example, in Second Life, a 3-D virtual world entirely built and owned by its residents. Since opening to the public in 2003, it has grown explosively and was inhabited by a total of 7, 965, 038 residents from around the globe, on July 11, 2007 -<http://secondlife.com/>

⁴ It has been commented that a bad bridge player is one who makes the same mistakes repeatedly; a good player is one who learns from her mistakes; and an expert is one who learns from the mistakes of other people.

⁵ What is feasible can only be decided when the actions are attempted in the real world. A more accurate model would help an

agent to estimate this feasibility more accurately.

- 6 There have been anecdotes from Japan of humans becoming emotionally bonded to virtual pets like Tamagotchi, rushing home to feed them, and feeling real sadness if they were to die. Web based virtual pets are popular with children. See <http://www.adoptme.com/>, <http://www.marapets.com/>, and www.neopets.com/. It has also been narrated by Pamela McCorduck (McCorduck 1973) that in the age of moving statues in medieval Europe, people were prone to ascribe humanlike thinking qualities to statues that could nod or shake their heads on being asked a question.
- 7 The term comes from Philosophy, and it means the study of being or existence. It is a study of notions of reality. A quest for answering questions about what the world is about, and questions about ourselves in this world. Computer scientists adopted the term when they wanted to represent, and reason with, such knowledge, particularly in the context where programs need to “talk” to each other and exchange such information; a situation which has become more and more prevalent with the advent of the internet and the worldwide web.
- 8 Bugs Bunny is a popular carrot chewing animated rabbit created by Warner Brothers.
- 9 Some people argue that the kind of learning done in artificial neural networks is similar to our acquisition of procedural knowledge that cannot be articulated.
- 10 But what *is* reality? We only know the world as it exists in our heads. Physicists have been splitting matter for more than a century without yet being able to construct a theory of the universe.
- 11 Buddhist monks too have long asserted otherwise. Modern psychology also says that we can control the pace of *our* time (see for example (Mansfield, 1998)).
- 12 In Chennai, where the author lives, we do not need such wizardry, unless you are interested in the subtle distinction between hot and hotter weather.
- 13 Even in Chennai.

agent to act in a *meaningful* manner, it must have some kind of *symbolic representation* about the world. To think, to contemplate, to cogitate about some real thing or event is to create some representation of it, and perhaps manipulate those representations. *The physical symbol system hypothesis* (PSSH) proposed by Allen Newell and Herbert Simon (1963) says that the ability to represent symbolic structures and manipulate them according to some fixed “laws” is *necessary* and *sufficient* to create intelligence. We will not argue about whether this is true, but it certainly seems to be a necessary perquisite for intelligent activity. By intelligent activity we mean activity that is goal-oriented, which can be achieved for example by “mindless” competition and survival, but sometimes requires a certain awareness of goals, awareness of the situation, and informed decision making. The only way one can act intelligently, or profitably, is by being able to *imagine*, or create imaginary imitations of, the real thing. Otherwise, the activity does not exist in reality.

Imagination is the key to intelligence. Imagination happens in the *mind* of the thinker, while action happens in the *world*. Imagination is necessary for awareness. It is sufficient if the premises are true for the conclusion to be true. It is necessary for the premises to be true for the conclusion to be true. Contrariwise, it is necessary for the conclusion to be true for the premises to be true.

The essence of *logical reasoning* is (a *mechanism* of) arriving at incontrovertible inferences in some representation of the world.

Formal Logic

Formal logic is the machinery for realizing such reasoning. It is essentially a symbol manipulation machine¹. In formal logic, statements taken to be true, the machinery determines what other sentences can be argued to be true. The truth or validity of an argument (inference) depends only on the *form* of the argument. The validity of an argument does not depend upon the *content*, or what is being said in the arguments. Consider the two arguments below. The first of which is the famous Socratic argument put forth by Aristotle (384 BC-322 BC).

Premise 1:	"All men are mortal"
Premise 2:	"Socrates is a man"
Conclusion:	"Socrates is mortal"

The following argument has an identical form, but obviously different content.

Premise 1:	"All soccer stars are rich"
Premise 2:	"Steven is a soccer star"
Conclusion:	"Steven is rich"

The form of both the arguments is identical, and both *arguments* are valid. Whether the conclusions of the arguments are valid (true) or not, depends upon the premises in the argument. Given a valid argument, if the premises are true, the conclusions are necessarily true. But if even one of the premises is not true it is not necessary for the conclusion to be true. For example if the first statement in the second argument (All soccer stars are rich) were false, then it could be possible that the conclusion (Steven is rich) is false. If it is not the case that all soccer stars are rich, Steven could well be one of those who is not. At the same time it is not necessary for the premises to be true for the conclusion to be true. Which means that the conclusion (Steven is rich) could still be true even if the premises (All Soccer stars are rich) is false. The fact that the premise is false does not mean that no soccer star is rich.

It is sufficient if the premises are true for the conclusion to be true. It is not necessary for the premises to be true for the conclusion to be true.

Entailment

One way all humans have intrinsic curiosity, a desire to know the truth about something or the other. They rely upon our senses. Another source is logic². In logic one is usually interested in dealing with true statements³. Which means that given a collection of true statements (premises), one is interested in knowing what other statements are logically *entailed* (necessarily made true) by the premises. Figure 12.1 below gives a set of premises and the sentences entailed by them.

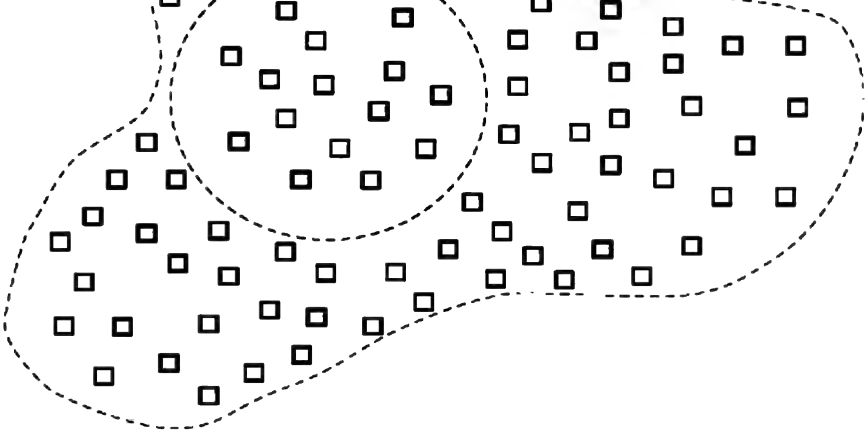


Figure 2.1 A schematic view of entailment. Each box represents a true statement. The boxes in the shaded region represent premises. The boxes outside the shaded region represent statements that are true as a consequence of the premises.

The task of reasoning in logic is to determine the set of statements that are entailed by a given set of statements (premises). Let S be the set of premises that we take for granted to be true. The task is to find the set T that contains all statements that are necessarily true as a consequence of the set S . Then S is the set of all statements that are true, and the set T is the set of all true statements including the ones that are implicitly true. Both sets S and T pertain to sentences in a given language.

Proofs

Logic is concerned with true statements. Determining whether a statement is true or not may be a difficult task. Logic adopts a different route to arriving at true statements. One that involves the entirely new concept of a *proof*. A proof is made up of a sequence of *inference* steps. Each inference step allows one to add a new statement to the existing set of sentences. Each inference step is based on a *rule of inference*. The rule used in the above two arguments is the *syllogism*, and can be expressed as follows.

Premise 1:	All X's are Y's
Premise 2:	M is X
Conclusion:	M is Y

The rule says that if one has (instances of) the first two statements in a set, then one can add the (consequent of the) third statement to the set. It is purely a syntactic process. We will say that given the antecedent, the rule produces the consequent. There could be many rules of inference in a logic machine, as we shall see later.

A set of premises S and the set of rules R together determine the set of statements that can be added to S to form the set P of *provable* statements. The set P may be computed as follows.

```

AllRules( $S$  : premises,  $R$  : rules)
1    $P \leftarrow S$ 
2   while a new rule  $r \in R$  is applicable
3       do Let  $C$  be the consequent of  $r$ 
4            $P \leftarrow P \cup \{C\}$ 
5   return  $P$ 

```

Figure 2.2 A simple procedure to construct the set of all provable statements. It applies a rule and adds its consequent to the set, until no more rules are applicable.

In some domains the set P could be infinite. This means that the above procedure may never end. In practice however one is not always interested in computing the entire set P , but rather in answering the question: does a specific statement p belong to P or not. The following variation of the above procedure achieves this.

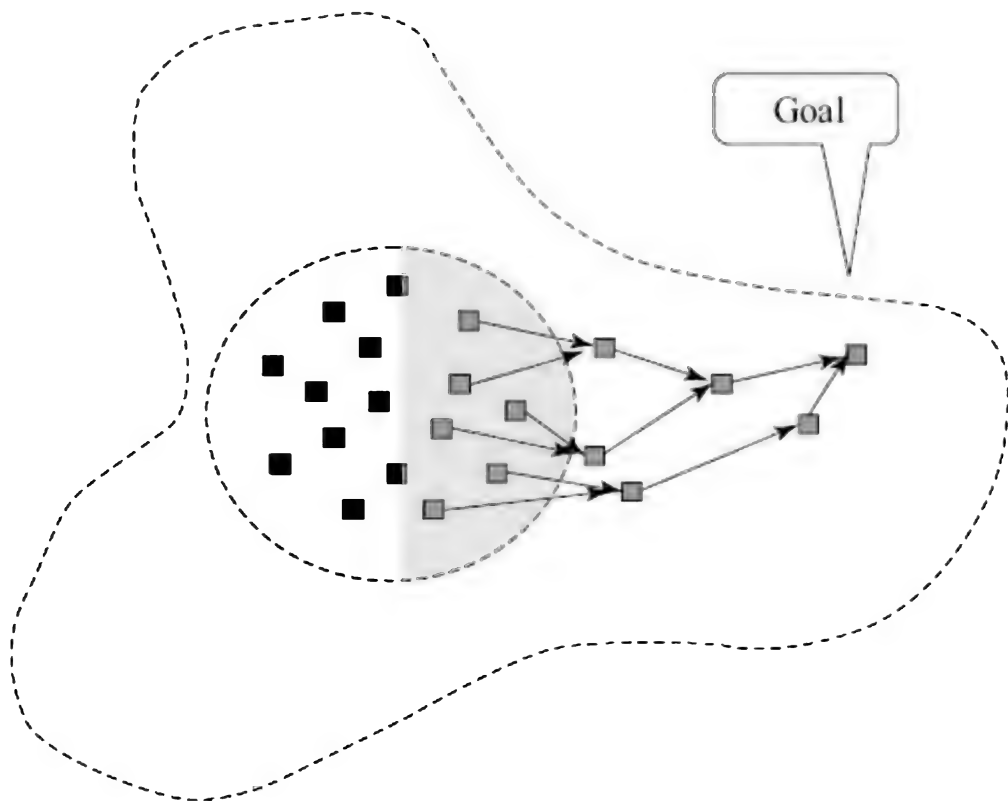
```

5         then return FAIL
6     Let C be the consequent of r
7      $P \leftarrow P \cup \{C\}$ 
8 until  $g \in P$ 
9 return TRUE

```

3 A simple procedure to test whether a given sentence can be generated by applying rules of inference.

observant reader would have noticed the similarity between this procedure and the *Simple-Search-1* in Chapter 2. Like in the search algorithms, the crux of the matter is in making the right choices of rules to be applied. Mathematicians, the community most concerned with proofs, are constantly looking for short and elegant proofs even of known theorems, when they are not trying to prove newer ones⁴. Another similarity between *Simple-Search-1* and the algorithm *SimpleProof* is that the algorithm *SimpleProof* does not return the solution. As shown in the figure, the solution is a tree rooted at the statement we want to prove (the Goal) whose leaves are statements in the set of premises S.

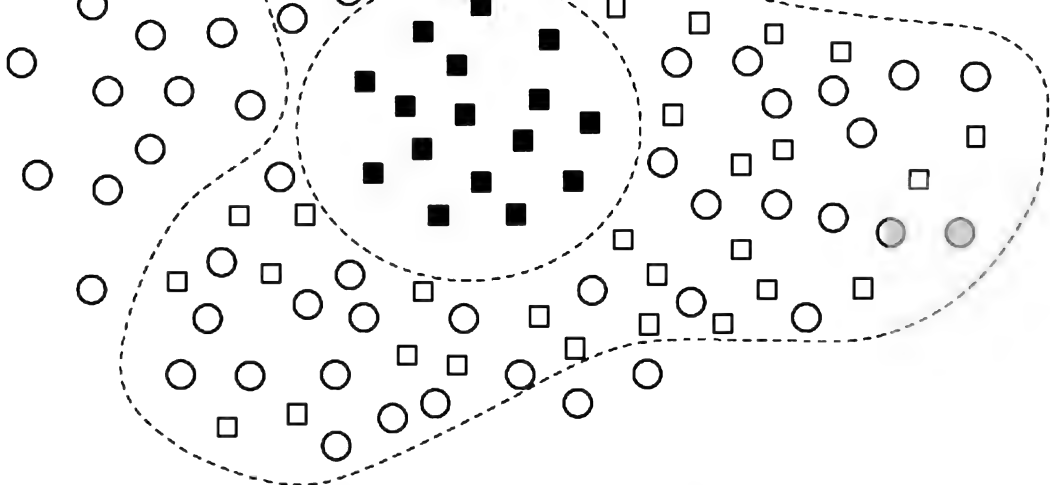


4 Finding a proof involves a sequence of inferences. In each inference a new sentence is added to the set. The procedure terminates when the sentence (Goal) is produced. The proof of the Goal is the tree as shown in the figure.

When searching for a proof a mathematician might have produced a lot of other statements, which are not part of the proof (tree) and have been discarded. If one were to automate the procedure then obviously one would face the question as to which rule to apply to what data at each inference step.

One can think of the task of finding a proof as a one player game, in which the player has to find a way to reach the goal on the board. Each rule of inference dictates what tile can be placed under what conditions. The condition for placing a particular tile may be achieved by placing some other tiles. Since the task is to place the goal tile, a goal reasoning approach akin to the Goal trees described in Chapter 6 suggests itself. It is in fact a wide area of research and we will look at it later in the chapter.

Soundness and Completeness



5 The set of provable statements is the set of premises plus the statements that can be produced by repeated application of the rules of inference. The set of true statements is represented by the shaded circles here. The two sets P and T may be different, as illustrated here. The empty boxes represent true statements that are not provable (are not in P). The circles outside the outer boundary represent provable statements that are not true (do not belong to T).

For a logic machine to be appropriate for our task of finding (at least in principle) only and all true statements, the set of provable statements P and the set of true statements T must be identical. The machine must produce only true statements, and it must be able to produce all true statements. It must not produce any false statements. As a corollary given any statement, it must produce either the statement or its negation, whichever is true, but not both. These properties have been described as soundness, completeness, and consistency.

Soundness: A logic is said to be sound if it produces only true statements. Or in other words, it does not produce any false statements. Formally a logic is sound if and only if P is a subset of T . This can be ensured by choosing the rules of inference judiciously (discussed later).

Completeness: A logic is complete if it produces all true statements. Formally, a logic is complete if and only if T is a subset of P . This can be ensured, when feasible, by choosing a sufficiently large number of premises and rules of inference.

Consistency: A logic is consistent if it does not produce both a sentence and its negation. This is a consistency property. A logic being sound.

It is important to note that a logic that does not produce *any* statements is by definition sound (but not complete). Likewise, a logic that produces *all* statements is complete (but not sound). We need a logic machine that is both sound and complete. If we can construct such a machine then we can rely on it entirely to decide whether a given statement is true or false. If it is true it will produce it, and if it is false it will halt without producing it (provided it is decidable).

In the above paragraph we have used the term "all statements". We need to clarify what we mean by that. Associated with every logic machine is a *language* that is used to express the statements in the logic. The language is defined over an alphabet along with syntax rules to determine whether a string on the alphabet is a sentence or not. We also use the term *well formed formula* or simply *formula* to refer to a sentence in logic. Given a language and the syntax we can determine the set of all formulas F that can be expressed in the language. The set of all statements that *can* be made using that language.

The concepts of completeness and soundness are also defined over this set. Completeness of a logic then means that it can produce every sentence in F that is *true*.

The properties of a logic machine are determined by its language or syntax. The more expressive the language is, the more difficult is it to produce all true statements. It is easy to build sound, complete and consistent logics in the simplest logic, known as *Propositional Logic*. But as we move higher to the more expressive logics such as *First Order Logic (FOL)* we lose the property of decidability. While first order systems are complete, for an input formula or sentence that is false, the procedure may loop for ever. If we further move to more expressive *second order logic* we even lose the property of completeness.

The logics named above are often classified as *classical* logics. By this we mean two valued logics in which each statement is assigned a truth value only once, exemplified by the domain of mathematics. A statement in mathematics is either true or false, and remains so ever after. Logicians have worked on variations on classical logic for change and for uncertainty.

elves. If they do so then would they have an understanding of what they are doing? We will not try and answer here.

History of Logic and Knowledge

As we move on to formal logic as a language for knowledge representation and theorem proving technologies, we take a quick look at the evolution of logic over the times. The reader not interested in the tales from the past can skip on to the section on Propositional Logic.

The Last Millennium

The notion of the mind did not exist in our minds always. And it probably does not apply to many life forms. We can postulate or argue for or against the idea of a frog, or a lizard, or a mosquito having a mind⁵. But we *will* argue about the notion of a mind with the representation and manipulation of symbol systems. John Haugeland (1985) in *Intelligence: The Very Idea* gives an insightful account of the emergence of the notion of the mind as a thought.

One of the first realizations that the world as in the mind may be distinct from the “real” external world was the publication of the geocentric model of the universe by Nicolaus Copernicus (1543). The mental model of the stars went around the earth, turned out to be different from the real phenomenon of the earth rotating around the sun, an illusion of the motion of the heavenly bodies. Galileo Galilei (1564–1642) showed that one can create a model by reasoning with geometric representations. He also believed that the sounds we hear, and the smells we perceive, are created in reality by tiny particles which had no direct relation to the smell or sound we *perceive*. The English philosopher Thomas Hobbes (1588–1679), who Haugeland calls the “The Grandfather of modern thinking as manipulation of symbols. Hobbes believed that thinking was like talking, only that it happened in the mind.

Thinking or the use of language is essentially a symbol manipulation activity. A sentence like “Akira ate an apple” is only a symbolic representation which the reader has to interpret. Each word is a symbol structure that stands for, or stands for, something else. The same can be said in other symbol systems “Akira aß einen Apfel”, “Akira saeb khaya”, or “Akira comió una manzana”. Each word holds some (hopefully shared) meaning for the speaker and the listener. And our processing of language is based on the understanding of that shared meaning. One could process symbols without understanding the intended meaning, just like the carrier of a message.

A computer is basically a symbol processing unit, at the heart of which two specific voltage levels stand for zero and one. All other sophisticated applications, like the word processor on which this document is written, use compound structures made up of ones and zeros to symbolize more complex concepts, and process them without understanding what they stand for. An interesting issue about “intelligence” was raised by the philosopher Searle, in the “Chinese room thought experiment” (Searle, 1980), in which he questions whether a system processing symbols without understanding the meaning behind them can be intelligent⁶. From our experience, we know that when a child learns to add two numbers, the procedure actually learnt is a symbol manipulation procedure (add the least significant digits, take the carry, and so on). Its association with the real world numbers has to be learnt separately. Many older students learn to solve complex mathematical equations without the foggiest idea of what is going on behind the symbol manipulation. But they do follow the rules that produce correct and useful answers.

However, while Hobbes did postulate that thought was like manipulation of mental particles the question still perplexed him (and still to a considerable extent perplexes us). The celebrated Rene Descartes observed that the symbol and the symbolized are two entirely different things. One can talk of a symbol manipulating system, but it could symbolize anything in the real world. The notion that symbols (thoughts) exist in different media leads to the problem of *mind-body dualism*. If what goes on in the mind is fundamentally different from what goes on in the physical world then how do the two interact? If one *thinks* of drinking some water, how does the hand move to accomplish that? How can matter interact with thought? a question that never was resolved, though several arguments were presented. One had to add the notion of “mechanical reason”: if something is mechanical it cannot reason, and if it can reason it cannot be mechanical. If reasoning is the *meaningful* manipulation of symbols then who is the manipulator? Descartes was in the ridiculous that there had to be a *homunculus* or a little man inside the head to manipulate the symbols. One could have realized that even if this were to be an answer, then it leads to a problem of infinite regress.

of human thought (an idea due to Descartes) in which each fundamental concept would be represented by a "real" character. "It is obvious that if we could find characters or signs suited for expressing all our thoughts, and as exactly as arithmetic expresses numbers or geometry expresses lines, we could do in all that we are subject to reasoning all that we can do in arithmetic and geometry. For all investigations of truth upon reasoning would be carried out by transposing these characters and by a species of calculus." (Principia Philosophiae, 1677. Revision of Rutherford's translation in (Jolley, 1995))

It is the relevance of the above statement made more than three centuries ago for artificial intelligence that motivates the physical symbol system hypothesis. The principles of Leibniz's logic are the following:

1. All our ideas are compounded from a very small number of simple ideas⁸, which form the alphabet of thought.

2. Complex ideas proceed from these simple ideas by a uniform and symmetrical combination, analogous to arithmetical multiplication.

Leibniz believed that complex thoughts would be represented by combining characters for simpler thoughts. He proposed assigning characteristic numbers to concepts. For example, basic concepts are assigned prime numbers and complex concepts non-primes, so that the composition of the complex concept out of basic concepts is revealed by the multiplication of primes giving a non-prime characteristic. Leibniz gives an example in the "Arithmetica Calculi" of 1679, where he assigns 2 to "animal" and 3 to "rational". He concludes that "humans" are characterized by 2×3 , i.e. 6, as humans are rational animals. In a similar fashion, he argues that an ape has a characteristic 10 is not a human, since neither is 10 divisible by 6 nor 6 by 10. But both have 2, i.e. being an animal (Schroeder 1997). Inspired by the mechanical calculators built by Blaise Pascal (1623–1662) he was interested in reasoning machine, the *calculus ratiocinator*, which current day AI researchers would recognize as a computer. Leibniz was a person of diverse interests and among other things he invented calculus and the binary system.

A concise history of logic can be found in (King and Shapiro, 1995). By the time George Boole and Augustus De Morgan (1806–1871) came onto the scene the system of using explicit variables to stand for things was already in use. A large number of philosophers had contributed to the development of logical reasoning. De Morgan discovered some of the basic laws of propositional logic⁹. But it was Boole who clearly stated the principles of Boolean algebra. "As to the lawfulness of this mode of procedure, it may be remarked as a general principle that it is not of the peculiar language of Mathematics alone, that we are permitted to employ symbols, but that whatever we choose that they should represent—things, operations, relations, etc., provided 1st, that the signification once fixed, 2nd, that we employ the symbols in subjection to the laws of the things they represent." (MacHale, 1985).

Charles Peirce was interested in developing a calculus for reasoning with an orientation of abstract algebra. His work is what modern computer scientists call Boolean algebra, which is used for reasoning about logic circuits. Charles Peirce (1839–1914) showed that all logical formulas could be implemented by a single operator (Peirce arrow: NOT followed by stroke: NAND). He also did pioneering work in *semiotics*, the science of symbols, along with Leibniz. Peirce's thought focused on laying the logical foundations of different branches of mathematics. These include Giuseppe Peano (1858–1932: number systems), David Hilbert (1862–1943 : geometry), and Ernst Zermelo (1871–1953: axiomatization of set theory). In 1920 Hilbert launched an ambitious program for formalizing mathematics, the aim of which was to lay a solid logic foundation of all mathematics (see (Detlefsen 1993)). In September 1930 the mathematician Kurt Gödel (1906–1978) put an end to that dream by proving that it is impossible to formalize all mathematics in a consistent manner. Gödel's Incompleteness Theorem (Gödel, 1931) had a lasting impact on logic and philosophy. It has also been raised when debating the possibility of formalizing the process, because the process of implementing anything on the computer is a form of formalization.

We are however at the moment interested in a different branch of evolution of logic that was concerned with the logic of scientific or rational *discourse*. This branch is characterized by the works of Bertrand Russell and Ludwig Wittgenstein (1889–1951), and Gottlob Frege (1848–1925). Of these Frege is the one credited with the foundation of formal logic as we know it now, and Wittgenstein and Russell (Whitehead and Russell) developed it further. It was Frege who introduced the notion of quantified variables and complex terms, and the notion of "proof" (see (Zalta, 2005) for an account of the works of Frege). With his system of logic we can represent and distinguish the sentences "every boy loves a girl" and "a girl is loved by every boy". The nesting of quantifiers is possible in his formalization, for example "every boy who loves a girl who is a friend of some of his classmates even who are poor in mathematics is a friend of some boy who hates history". The power of logic is that one could reason over a collection of such sentences, and that is the power of logic we seek to formalize.

It will be true in all possible worlds, while a possibly true statement will be true in some possible worlds. This applies to what is known as *Intuitionistic Logic*, which has a stronger notion of truth allowing only constructive proofs (see (Van Dalen, 2001)). In particular, it does not accept Aristotle's law of excluded middle that for any proposition P , either P is true or its negation $\neg P$ is true (and there is no middle ground)¹¹. Thus since $(P \vee \neg P)$ is a tautology, one can infer that one of the statements implies the other. That is $(P \supset Q) \vee (Q \supset P)$. Since P and Q can be anything, for example, "The Earth is round" and "Two plus two is four", the connection between them is not intuitively acceptable. This is because neither in fact causally implies the other. Another variation is *Default Logic*, which allows the expression of a default implication, like between clouds and rain, is *Default Logic*. These are true in general but not universal truths like in classical logic. We shall look at reasoning with default logic in chapter 17.

The semantics of logic *defines* the meaning of statements and determines when they are true. But logic also provides another route to the truth values of statements, via a *proof*. This is of great importance, because a proof is that is independent of meaning. It is entirely syntactic in nature. An important consequence of this is that we can write programs to generate proofs, without having to worry about the "computer knowing" the meaning of the statements it is manipulating. It is necessary to build proof procedures that are *sound* to guarantee that the statements derived by the proof procedure are true. A related property of a logic machine (system) is whether it is *complete*. Completeness means that all true statements that can be expressed in the language will have proofs. The work of Frege, Russell, Hilbert and Gödel resulted in the development of proof systems. This was extended by Gerhard Gentzen (1909–1945) who devised the natural deduction system and the sequent calculus.

Logic in Ancient Greece

The notion of formal logic that evolved in Europe can be traced back to Aristotle (384–322 BC), who was a philosopher (384–322 BC)¹², who himself was a disciple of Socrates (469–399 BC). Aristotle made significant contributions to physics, natural history, psychology and philosophy. The roots of western rationalism can be traced back to the Socratic method (or Method of Elenchus) in which one examines the possible counterpoints to one's theory, asks questions and eliminates them to arrive at the conclusion. The method was described in a series of dialogues known as the Socratic dialogues written by Plato and Xenophon (431–355 BC). The dialogues are presented as discussions between Socrates and other persons of his time, or as discussions between Socrates' students. Plato's *Phaedo* is an example of this latter category (see for example (Plato and Fowler, 1997)). The formalization of the reasoning process is described in Aristotle's *Organon* (meaning *Instrument*). The *Organon* is the name given by Aristotle's followers, the Peripatetics, to the standard collection of his six works on logic¹³.

Categories: Aristotle's 10-fold classification of that which exists. These categories consist of substance, quantity, quality, relation, place, time, situation, condition, action, and passion.

On Interpretation: Aristotle's conception of proposition and judgment, and the various relations between them: affirmative, negative, universal and particular propositions.

Prior Analytics: On what can be taken to be true by itself, the syllogisms.

Posterior Analytics: deals with demonstration, definition, and scientific knowledge. A demonstration is a syllogism that leads to a proof.

Topics: Treats issues in constructing valid arguments, and inference that is probable, rather than certain.

On Sophistical Refutations: gives a treatment of logical fallacies, and provides a key link to Aristotle's *Metaphysics*.

Aristotle was largely concerned with *deductive arguments*, as are we, and he defines deduction as follows: "Deduction is speech (logos) in which, certain things having been supposed, something different from these results of necessity because of their being so." (Prior Analytics I.2, 24b18–20)

What the "things supposed" is a premise of the argument, and what "results of necessity" is the conclusion. A syllogism was concerned with four kinds of statements,

: universal affirmative	(for example, "All men are mortal")
: universal negative	(for example, "No man is immortal")
: particular affirmative	(for example, "Some men are bright")
: particular negative	(for example, "Some men are not bright")

Statements can also be categorized as *Asp*, *Esp*, *Isp*, and *Osp*, where "s" stands for subject and "p" for predicate.

that was not directly obtained (through sensuous means). While many schools of thought emerged, the skeptics who refused to accept any claim of knowledge. One of the more famous ones, Samkhya, whether there was afterlife, said "I do not say there is an afterlife and I do not say there is no afterlife" (Radhakrishnan, 1997, 2000).

Indian philosophy goes back to the art of debating. The six well known schools of philosophy are: *Sāṃkhya* (around 500 BC), *Mīmāṃsā* (Jaimini around 300 BC), *Nyāya* (Akṣapāda Gautama, 2nd century BC), *Yoga* (Patanjali, 2nd century BC or later), *Vaiśeṣika* (Kanada, 6th century BC), and *Vedānta* (also called *uttara Mīmāṃsā*). *Vedānta* means the end of all knowledge, were composed starting the 9th century BC in the *Upaniṣads*. *Vedānta* is based on Veda Vyāsa, also the author of the Indian epic, *Mahābhārata*). Of these the *Nyāya-Vaiśeṣika* is the most logical system, based on logic and argumentation. The *Nyāyasūtra* by Gautama (or Gotama) written in second century BC was concerned with the theory of knowledge of sixteen categories (Sinha and Vidyabhusana, 1930)¹⁵:

Means of valid knowledge (*pramāṇa*),
Objects of valid knowledge (*prameya*),
Doubt (*saṃśaya*),
Purpose (*prayojana*),
Example (*dṛṣṭānta*),
Conclusion (*siddhānta*),
The constituents of a syllogism (*avayava*),
Argumentation (*tarka*),
Ascertainment (*nimāṇa*),
Debate (*vāda*),
Disputations (*jālpa*),
Destructive criticism (*vitanda*),
Fallacy (*hetvābhāsa*),
Flimsiness (*chala*),
Refutations (*jāti*), and
Points of the opponent's defeat (*nigrahasthāna*).

The four possible sources of knowledge—perception (*pratyakṣa*), inference (*anumāna*), comparison (*upamāna*), and verbal testimony (*śabda*, which could be of God from the *vedās*, or of a trustworthy human being). *Anumāna* is concerned with logical necessity.

is reflected in the form of the argument now known as the five step syllogism. Suppose that we see smoke on a mountain, and we want to infer that the mountain is on fire. The *Nyāyasūtra* describes the process of inference as a five step process (Sinha and Vidyabhusana, 1930) of a good argument as a five step process.

First, a statement of the thesis (*Pratijñā*): there is fire on the mountain
Second, a statement of reason (*Hetu*): because there is smoke on the mountain
Third, an example of the underlying rule (*Udahāraṇa*): where there is smoke there is fire, like the mountain in the forest
Fourth, a statement that (*Upānaya*): this case is like that
Finally the assertion of the thesis proven (*Nigamaṇa*): therefore the mountain is on fire

The derived piece of knowledge is known as *anumāna* (after cognition). This is in contrast to the three step syllogism exemplified by the Socratic argument. It has also been observed (Radhakrishnan, 1953; 1959) that the Indian philosophers used the five step reasoning only when the task was to prove their conclusions. When the task was to infer something for oneself the simpler three step process followed.

There is smoke on the mountain.
Wherever there is smoke there is fire.
Therefore, there is fire on the mountain.

This is precisely the form of reasoning, the Aristotelian syllogism, which is fundamental to western logic. This was basically a part of the theory of knowledge (*Pramāṇa Sāstra*). Indian philosophers were also concerned with the validity or truth of each statement, and asked how it could be true. Some statements were ascribed to be true because they were directly perceptible to be true¹⁶, such as there is smoke on the mountain.

vyāpti must admit at least one instance. This is different from modern western logic which is based with knowledge about some real situation, but only with the structure of sound argument. Thus a statement like “all three legged fishes have one eye” in logic as true without any problems because there are no three legged fishes. When logic and reasoning is part of a complete knowledge system, a rule must be supported by a positive example, or a negative example, or by both a positive and negative example (Müller, 1853). He says that “Everything that is knowable is nameable”, one can give only positive examples like a jar. One can then use it to infer that a jar is nameable because it is knowable. If one wants to infer that a jar is nameable from all the other four elements¹⁷, because it has odour” it does not make sense to use a rule like “a jar is nameable from other elements has odour” because the *only* thing that satisfies the above case is earth. One can use a negative *vyāpti* and say that “whatever is not different from the other elements, has no odour”. One can add an instance like water, or light. The example of smoke and fire we have been looking at admits both positive and negative instances. We can give a positive example, the culinary hearth. And also an instance of a negative version “wherever there is no fire, there is no smoke, as in a lake”.

vyāpti characterizes the universal rule by identifying the two components as *linga* the symbol for the reason and *linga* the symbol for the conclusion. He says that for the *vyāpti* to hold all occurrences of the *linga* must be accompanied by the *sādhya*, also that the *linga* must not occur without the *sādhya*. His successor two centuries later, Śaṅkara, categorized the *vyāpti* into two kinds. One, where the *linga* has the “own nature” of the *sādhya*. This is the case of the latter, for example a Siberian tiger is a kind of a tiger, and thus one can say that whenever there is a tiger, X is a tiger. These kind of taxonomic relations are common in all sciences now. We shall return to these in more detail later. The other kind of relation that holds is a causal connection between the *linga* and the *sādhya*, as between smoke and fire. The Naiyāyika philosophers again generalized the condition for a *vyāpti* to be a causal connection. “X is absent from all those instances from which Y is”. This says the X can only occur if Y occurs. So, if there is no fire, there is no smoke. Wherever there is no fire.

Modern logic has largely been subservient to philosophy. It has been preoccupied with the nature of existence and what can accept to be true.

Modern logic has its roots in a system developed by Aristotle. It is a tool that one can use for making valid inferences. It is that *given* a set of premises or statements accepted to be true, to decide what other statements can be inferred or are entailed.

It is fine for the use of computers for theorem proving and reasoning, because we expect them to work according to the rules we define for them. But if at a later stage we would want our machines to philosophize then we need to be able to be concerned with what is really true in the domain of discourse, and then look into why something is true.

Now come back to our study of logic as a mechanism for reasoning.

Propositional Logic

Propositional logic is basically symbol processing machinery. A logic machine is constructed by defining a language, and then adding new sentences. The language is a formal language in which well formed formulas represent sentences. We begin by studying the simplest language, the propositional logic. The language for propositional logic consists of two parts.

The first part is the logical part, which is independent of what the symbols are supposed to symbolize. The second part is the non-logical part which is concerned with what is being said.

The logical part of the vocabulary contains symbols like “ \wedge ”, “ \vee ”, “ \neg ”, and “ \supset ”, read as “and”, “or”, “not”, and “implies”, respectively. It also includes sets of brackets “(”, “)”, “[”, “]”, “{”, and “}”. The symbols “ \perp ” and “ \top ” read as “bottom” and “top”.

The non-logical symbols in propositional logic consists of a countable set of symbols $\mathbb{P} = \{P_1, P_2, P_3, \dots\}$. These symbols $\{P, Q, R, \dots\}$. These symbols stand for atomic sentences or atomic formulas or propositional variables. A sentence or formula is the smallest unit to which a truth value can be assigned.

We will use the symbols α , β , γ , and δ to denote sentences (or formulas) in the language. The set of sentences in the language is denoted by the following,

$$\perp \in S_p$$

$$\top \in S_p$$

$$\text{if } \alpha \in S_p \text{ then } \neg \alpha \in S_p$$

Propositional Logic Semantics

meaning of an atomic sentence $\alpha \in \mathbb{P}$ is decided externally by the user. It may stand for any statement like “The sky is blue” or “The mountain is on fire”. The *truth value* of an atomic sentence is also decided externally. A valuation function maps all atomic sentences to the set $\{true, false\}$, or equivalently $\{t, f\}$, or $\{\top, \perp\}$ or $\{1, 0\}$. A valuation function, V , is a function that maps atomic sentences to the set $\{true, false\}$.

$$V: \mathbb{P} \rightarrow \{true, false\}$$

A natural language sentence that can in principle be mapped to *true* or *false* is a proposition. This is like “White can always win in chess” or “Chess games are always drawn with perfect play”, even though we may not know if they are true or not. Propositions are assertive statements, or assertions. Interrogative sentences like “Do you want to play chess today?” or imperative sentences like “Let us play a game of chess” are not propositions. As Russell showed, certain self-referential sentences that appear to be assertions are also to be excluded. These are sentences like “This sentence is false” or even a combination of sentences like “The following sentence is true if and only if the preceding sentence is false.” Such sentences are at the heart of Gödel’s Incompleteness theorem. The meaning and the truth value of a compound sentence are determined by the meaning of the component sentences and the logical connectives. The meaning of logical connectives is fixed. The truth value of compound sentences, the mapping $\text{Val}: S_P \rightarrow \{true, false\}$ is determined by structural induction as follows,

$$\text{Val}(\top) = \text{true}$$

$$\text{Val}(\perp) = \text{false}$$

$$\text{For all } \alpha \in \mathbb{P}, \text{Val}(\alpha) = V(\alpha)$$

$$\text{If } \text{Val}(\alpha) = \text{true} \text{ then } \text{Val}(\neg\alpha) = \text{false}$$

$$\text{If } \text{Val}(\alpha) = \text{false} \text{ then } \text{Val}(\neg\alpha) = \text{true}$$

$$\text{If } \text{Val}(\alpha) = \text{false} \text{ and } \text{Val}(\beta) = \text{false} \text{ then } \text{Val}(\alpha \vee \beta) = \text{false} \text{ else } \text{Val}(\alpha \vee \beta) = \text{true}$$

$$\text{If } \text{Val}(\alpha) = \text{true} \text{ and } \text{Val}(\beta) = \text{false} \text{ then } \text{Val}(\alpha \supset \beta) = \text{false} \text{ else } \text{Val}(\alpha \supset \beta) = \text{true}$$

$$\text{If } \text{Val}(\alpha) = \text{true} \text{ and } \text{Val}(\beta) = \text{true} \text{ then } \text{Val}(\alpha \wedge \beta) = \text{true} \text{ else } \text{Val}(\alpha \wedge \beta) = \text{false}$$

Validity, Satisfiability and Unsatisfiability

A sentence or formula in logic falls into one of the following three classes.

Valid formulas. There are some formulas that are true independent of the valuation of their atomic components. Such a formula is called a *valid* formula or a *tautology*. The simplest example is $(P \vee \neg P)$ but many others exist, for example $((P \wedge Q) \supset R) \equiv (P \supset (Q \supset R))$.

Satisfiable formulas. Satisfiable formulas are those that can be made true by certain valuation functions. A formula is also known as a *contingent* formula. A simple example of this is $((P \wedge Q) \supset R)$, which is true for some valuations (amongst other satisfying valuations).

Unsatisfiable formulas. Unsatisfiable formulas or contradictions are formulas that cannot be made true by any valuation. The simplest example is $(P \wedge \neg P)$, but many more examples exist. In fact, for every formula α , its negation $\neg\alpha$ is unsatisfiable.

The validity (or unsatisfiability) of formulas can be evaluated by considering all possible valuation functions. For each atomic proposition can be mapped to one of two values, true or false, if there are N propositions, one will have to consider 2^N different valuation functions to determine whether the formula is valid (or satisfiable). This is what one does by constructing a *truth table*. For satisfiable formulas, on the other hand, if we can find *some* valuation that makes it true. One may not have to inspect the entire truth table. Various methods have commonly been used for solving the SAT problem as it is known (see Chapter 4). The validity of formulas, or its converse unsatisfiability of formulas, is of special interest because logic is the foundation of computer science. Logic is concerned with the notion of entailment. We illustrate this with an example. Consider the following set of sentences:

T = Alice will go to college.

U = Alice likes chemistry.

V = Alice likes history.

the given facts are,

$P \wedge Q$)

$P \supset R$)

$R \wedge S \supset T$)

$Q \vee S$)

$U \wedge \neg V$)

the goal or theorem to be proved is the proposition T . The sentences 1-5 are the *premises* and the sentence T is the *conclusion*. We say that the sentence T is entailed by the sentences 1-5 iff whenever the sentences 1-5 are true, the sentence T is true. In other words we want to say that the following formula is always true.

$\{[(P \wedge Q) \wedge (P \supset R) \wedge ((R \wedge S) \supset T) \wedge (\neg Q \vee S) \wedge (\neg U \wedge \neg V)] \supset T\}^{18}$

call the sentences 1-5 as the knowledge base KB then we write the the entailment relation as,

$KB \models T$

say that the sentence T is entailed by the knowledge base KB (Brachman and Levesque, 2004). We write this as,

$\models \{[(P \wedge Q \wedge (P \supset R) \wedge ((R \wedge S) \supset T) \wedge (\neg Q \vee S) \wedge (\neg U \wedge \neg V)] \supset T\}$

the fact that a tautology is entailed by nothing.

we observe that the sentence T can also be true when 1–5 are not all true, for example when sentence 5 is false. The sentences 1–5 being true is *sufficient* for T to be true, though it is not necessary for them to be true. On the other hand, it is *necessary* that sentence T be true for sentences 1–5 to be true. That is, T has to be true if sentences 1–5 are true.

In an implication $(\alpha \supset \beta)$ we say that β is a *necessary condition* for α ; and α is a *sufficient condition* for β .

One way to verify whether an implication is a tautology is by looking at *all* the valuations that make the premises true and checking whether the conclusion is true as well. If yes, then we can say that the conclusion is entailed by the premises.

If there are N propositions in the premises and conclusion put together this involves looking at 2^N valuations.¹⁹ This can be an expensive computation if the number of propositions is high. Instead logic uses a more efficient route to establish the truth of statements, the method of proof. In proof procedures the time complexity depends upon the complexity of the formula and not on the number of propositions used. As we will see in the next section, in restricted languages, the time complexity could even be polynomial.

We observe that the meaning or *content* of the sentence does not play *any* role in the validity of the argument. We can replace the sentences coded by the symbols P, Q, R, S, T, U, V with any natural language sentences and the argument will still hold. It is only the *form* of the argument that matters. That is why logic is known as *formal logic*.

Proofs

A proof procedure is a syntactic process. It is a symbol manipulation procedure that is oblivious of the semantics. We can think of it as a one person game in which there exists a set of tiles on a board, and the rules allow the player to move the tiles. Each tile stands for a sentence. The game starts with a set of tiles for the premises, and ends with a tile representing the conclusion is placed on the board.

If we think of the language of logic as the *object* language, then the rules of inference are expressed in a *meta* language. That is, the rules are outside the language, or are extra-logical. Each rule contains one or more *antecedents*, and one *consequent*. Each antecedent is a *pattern* that can match a sentence. The rule is applied if its antecedents has a matching sentence (in the existing set or “on the board”). When a rule is applied, the sentence described in the consequent is added to the set of sentences, or using the game metaphor, a tile is added to the board.

A sentence is provable if it can be *produced* by a sequence of rule applications on the knowledge base.

If a sentence α is provable we write this as,

$\frac{\alpha \vee \beta}{\beta}$ <p>Destructive Syllogism (DS)</p>	$\frac{\alpha}{\alpha \vee \beta}$ <p>Addition (A)</p>	$\frac{\alpha \wedge \beta}{\alpha}$ <p>Simplification (S)</p>
$\frac{\alpha \supset \beta \quad \beta \supset \gamma}{\alpha \supset \gamma}$ <p>Hypothetical Syllogism (HS)</p>	$\frac{\text{From } (\alpha \supset \beta) \wedge (\gamma \supset \delta) \text{ and } \alpha \vee \gamma}{\beta \vee \delta}$ <p>Constructive Dilemma (CD)</p>	$\frac{\text{From } (\alpha \supset \beta) \wedge (\gamma \supset \delta) \text{ and } \neg \beta \vee \neg \delta}{\neg \alpha \vee \neg \gamma}$ <p>Destructive Dilemma (DD)</p>

Figure 12.6 Some of the common rules of inference.

The most well known rule is called *modus ponens* (MP),

$$\frac{\alpha \supset \beta \quad \alpha}{\beta}$$

Figure 12.6 says that if there is a sentence α and a sentence $\alpha \supset \beta$ then we can add the sentence β to the set (K). A rule of inference connects true antecedents to true consequents. Then if one has a data set of true statements and some valid rules of inferences that are applicable, one can, by a purely syntactic process, produce new sentences that are guaranteed to be true. How does one decide that a rule of inference is sound? A rule is sound whenever it is based on a *tautological implication*, that is, a sentence in which the antecedent is the implication, and the sentence is always true. The underlying sentence for the rule *modus ponens* is

$$((\alpha \supset \beta) \wedge \alpha) \supset \beta$$

The following truth table that looks at all valuations of α and β , demonstrates that the sentence is indeed a tautology.

A tautological implication.

α	β	$(\alpha \supset \beta)$	$(\alpha \supset \beta) \wedge \alpha$	$((\alpha \supset \beta) \wedge \alpha) \supset \beta$
true	true	true	true	true
false	true	true	false	true
true	false	false	false	true
false	false	true	false	true

A rule of inference is an extra-logical statement that gives us license to add new sentences or formulas to a set of sentences. If the rule of inference is valid or sound, and one starts with a set of true sentences or premises, then the application of the rule will add more sentences that are true. If we build a logic system with sound rules of inference we build a *deductive* system. One can deduce new facts from old ones that are guaranteed to be true. We say that the logic is sound.

We can build a deductive logic system by choosing any rules that are based on a tautological inference. We are encouraged to verify that all the rules given in Figure 12.6 are sound. Let us look at an example of a rule that is sound.

$$\frac{\alpha \supset \beta \quad \alpha}{\beta}$$

first example is a familiar illustration of an erroneous inference that one might make. The second example, in the same form, is also not a sound inference. But as we all know such inferences are often made in detecting an illness, and can be very useful. However one must be careful to remember that such inferences may not be valid, which is why diagnoses are sometime wrong. We shall return to abductive inference later. We now on deductive inferences and constructing automatic theorem provers. A proof is a sequence of sentences (premises), a set of (sound) rules, and a desired sentence, a proof is a sequence of sentences culminating in the desired sentence. In a proof, each sentence is either a premise, or is generated by some rule. A proof for the above example is given below. Traditionally the justification of each step is given alongside.

1.	$(P \wedge Q)$	premise
2.	$(P \supset R)$	premise
3.	$((R \wedge S) \supset T)$	premise
4.	$(\neg Q \vee S)$	premise
5.	P	1, simplification
6.	Q	1, simplification
7.	R	2, 5, modus ponens
8.	S	4, 6, disjunctive syllogism ²⁰
9.	$(R \wedge S)$	7, 8, conjunction
10.	T	3, 9, modus ponens

This kind of a proof construction process is called natural deduction and owes its early development to Gerhard Gentzen. The reader would have noticed the similarity of the proof with a plan (Chapter 7). A rule of inference is an operator with only positive effects. The similarity extends to finding proofs as well. Some of the algorithms for planning can be applied to the finding proofs as well. In fact the backward search algorithms can be applied to finding proofs, because rules of inference, unlike planning operators, have only positive effects. The algorithms for problem decomposition can also be applied to finding proofs, both in the forward and the backward direction.

Consistency

It is important that one is dealing with a set of consistent statements. This means that both a statement and its negation should not be present in the set. This is because if both are present then any arbitrary statement can be “proved” as illustrated below.

1.	$P \wedge \neg P$	premise
2.	P	1, simplification
3.	$\neg P$	1, simplification
4.	$P \vee Z$	2, addition
5.	Z	4, 3, disjunctive syllogism

If Z is an arbitrary statement this would mean that our system is not sound.

Substitution

If inference allows one to add a new sentence to the database²¹. In contrast one can define a rule of inference that allows one to replace one sentence with another. This is possible when one sentence is logically equivalent to another. As an example, let us look at the following equivalence.

$$((\alpha \supset \beta) \equiv (\neg \alpha \vee \beta))$$

If the above equivalence is a tautology, then the sentence $(\alpha \supset \beta)$ will always take the same truth value as $(\neg \alpha \vee \beta)$. Hence, either of the two could be replaced by the other without any loss. We can verify if the above equivalence is a tautology by constructing a truth table.

ever one can construct a tautological equivalence one can define a valid rule of substitution. The picticts some of the commonly used rules of substitution.

$(\alpha \vee \alpha)$	idempotence of \vee
$(\alpha \wedge \alpha)$	idempotence of \wedge
$(\beta) \equiv (\beta \vee \alpha)$	commutativity of \vee
$(\beta) \equiv (\beta \wedge \alpha)$	commutativity of \wedge
$\vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma))$	associativity of \vee
$\wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma))$	associativity of \wedge
$\vee \beta) \equiv (\neg \alpha \wedge \neg \beta)$	DeMorgan's Law
$\wedge \beta) \equiv (\neg \vee \neg \beta)$	DeMorgan's Law
$(\beta \vee \gamma) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$	distributivity of \wedge o
$(\beta \wedge \gamma) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$	distributivity of \vee o
$(\text{true}) \equiv \text{true}$	
$(\text{false}) \equiv \alpha$	
$(\text{true}) \equiv \alpha$	
$(\text{false}) \equiv \text{false}$	
$(\neg \alpha) \equiv \text{false}$	
$(\neg \alpha) \equiv \text{true}$	
$\neg(\neg \alpha)$	
$(\beta) \equiv (\neg \beta \supset \neg \alpha)$	contrapositive
$(\beta) \equiv (\neg \alpha \vee \beta)$	implication
$(\beta) \equiv ((\alpha \supset \beta) \wedge (\beta \supset \alpha))$	equivalence
$\wedge \beta) \supset \gamma) \equiv (\alpha \supset (\beta \supset \gamma))$	exportation
$\supset \beta) \wedge (\alpha \supset \neg \beta) \equiv \neg \alpha$	absurdity

12.7 Some commonly used substitution rules. At any place in a sentence the left hand side may be replaced by the right hand side

Forward Reasoning

Following algorithm is a refinement of the procedure in Figure 12.3. The algorithm *SimpleProofProcedure* takes as input a set of premises S , a set of *instantiations* R of the rules²² applicable, and the goal statement. It searches in the forward direction till it produces the goal statement or no rules are applicable. At each stage it leaves a trail of the rules used, which is used to reconstruct the *Proof*.

```

    then return FAIL
    P ← P ∪ {consequent}
    applied ← Cons(r, applied)
return ReconstructProof(g, P, applied)

ReconstructProof(G : goal, S: premises, A : applied)
goalSet ← (G)
proof ← ()
while goalSet ≠ ()
do next ← Head(goalSet)
   goalSet ← Rest(goalSet)
   if next ∈ S
       then rule ← BackChain(next, A)
            antecedents ← Second(rule)
            goalSet ← Append(antecedents, goalSet)
            proof ← Cons((next, rule), proof)
       else proof ← Cons((next, "Premise" ), proof)
return proof

```

```

BackChain(C : consequent, A : rules applied)
if C = First(First(A))
then return First(A)
else return BackChain(C, Rest(A))

```

2.8 A simple forward-search proof procedure. Observe that we assume refactoriness: Each rule can only fire once with the same antecedents. This means that theorem proving does not backtrack. The *CHOOSE* operator makes the right choice non-deterministically. We assume that for each rule exist in the form (consequent, list-of-antecedents). The reversed Proof contains a sequence of statements along with a justification. The justification is either the rule used to produce that statement, or the string "Premise" if the statement is a premise.

In the above algorithm we have used the non-deterministic *CHOOSE* operator that somehow selects the right rule to apply at each inference step. In practice any of the search strategies we have used could be employed. Non-deterministic choice the search does not have to backtrack. This is because an inappropriate inference step does not produce a sentence that is not required for the proof. The set of sentences grows monotonically with each inference step. This has an impact though because the task of finding applicable rules becomes more expensive. Combined with the fact that some inferences may not be useful and only add to the computations, the choice of the correct rule to apply is what makes the complexity of the proof finding algorithm. This will become critical when we move on to theorem proving in first order logic.

Backward reasoning suffers from the same problem as forward search in planning. It lacks a sense of direction. A backward version of the above algorithm is likely to be much more focused. It will be simpler than the backward search algorithm because the problem of spurious states is not there. It may have to backtrack though if the consequent is produced by different rules. The reader is encouraged to write a backward version of the proof finding algorithm.

Rules vs Meta Rules

A rule of inference says that if the antecedents exist in the database then the consequent may be added to the database.

An implication $\alpha \supset \beta$ says that if α is true then so is β . In fact this is the property used in rule based systems (see Chapter 6).

We think of the rule of inference as an implication statement (antecedent \supset consequent) and add it to the database. Why do we need a separate rule of inference *outside* the set of sentences? Let us consider the rule of inference to answer that question.

is why we need rules of inference that are based on tautological implications but that are *not* based on inference. If we call $(\alpha \supset \beta)$ a rule (as in rule based systems) then we can call the rule of inference *modus ponens*. The meta-rule in fact sits in the inference engine, or the proof procedure that applies rules to produce proofs.

Resolution Method in Propositional Logic

As discussed earlier the completeness of logic systems depends upon the choice of premises (or axioms) and the rules of inference. A multitude of logic systems were devised in the 19th century. These systems varied on the choice of premises used in the logical part of the language, and on the rules of inference that were allowed. The task of automated proof finding (theorem proving) has to address the issue of which rule of inference to use and what data. The second issue becomes more critical in first order logic. The first one, of choosing a rule of inference, was effectively removed in 1965 when Alan Robinson invented the resolution refutation method (Robinson, 1965) because the resolution method requires only one rule of inference and is a complete method. We look at a proof procedure based on resolution refutation with propositional logic, and revisit it with first order logic.

The Resolution Rule

The resolution method requires that the formula is in conjunctive normal form (*CNF*). A formula F is in *CNF* if it is a conjunction of clauses.

$$F = C_1 \wedge C_2 \wedge \dots \wedge C_n$$

where C_i is the formula is a conjunction of clauses where each clause C_i is a disjunction of literals,

$$C_i = D_{i1} \vee D_{i2} \vee \dots \vee D_{ik(i)}$$

where $k(i)$ is the number of literals, and each literal is either a proposition or the negation of a proposition. Since the structure of the formula in *CNF* is known, it is also conventionally represented as a set of clauses.

$$F = \{\{D_{11}, D_{12}, \dots, D_{1k(1)}\}, \{D_{21}, D_{22}, \dots, D_{2k(2)}\}, \dots, \{D_{n1}, D_{n2}, \dots, D_{nk(n)}\}\}$$

A formula in propositional logic may be converted into *CNF* by the use of substitution rules based on logical equivalences. Conversion into *CNF* generally results in increase in the size of the formula, often a large number of clauses in the number of propositions.

The single rule used in the refutation method, called the resolution rule, takes two clauses that contain complementary literals as follows.

Clause 1:	$R_1 \vee R_2 \dots \vee R_k \vee Q$
Clause 2:	$P_1 \vee P_2 \dots \vee P_m \vee \neg Q$
Result:	$R_1 \vee R_2 \dots \vee R_k \vee P_1 \vee P_2 \dots \vee P_m$

where the literal Q and its negation $\neg Q$ are removed and the remaining literals are combined to form a new clause called the *resolvent*. We will use simpler clauses of two literals for illustration but the method applies to the general case as well. With the smaller clause the rule becomes,

Clause 1:	$R \vee Q$
Clause 2:	$P \vee \neg Q$
Result:	$R \vee P$

The validity of the resolution rule can be established by showing that adding the resolvent does not change the logical nature of the data. That is, the sets before and after are equivalent. It suffices to show that,

$$((R \vee Q) \wedge (P \vee \neg Q)) \equiv ((R \vee Q) \wedge (P \vee \neg Q) \wedge (R \vee P))$$

It is left as an exercise to show that the above is indeed a tautology. A consequence of this tautology is that the addition of the resolvent produced by the resolution rule does not change the logical nature of the data.

sequence of resolvents R_1, R_2, R_3, \dots culminating with \perp . The databases at all stages are logically equivalent, the resolution rule is sound.

$$\begin{aligned} \{P_1, P_2, \dots, P_N\} &\equiv \{P_1, P_2, \dots, P_N, R_1\} \\ &\equiv \{P_1, P_2, \dots, P_N, R_1, R_2\} \\ &\equiv \{P_1, P_2, \dots, P_N, R_1, R_2, R_3\} \\ &\equiv \{P_1, P_2, \dots, P_N, R_1, R_2, R_3, \dots, \perp\} \end{aligned}$$

since the last set of clauses evaluates to *false* (because it contains the empty clause) the set we started with, which is logically equivalent, also evaluates to *false*. Thus $\{P_1, P_2, \dots, P_N\}$ is *false*. The following simple pseudocode illustrates the resolution method in propositional logic.

```

PropositionalResolution(S : premises in clause form)
1  while TRUE
2      do  CHOOSE a new pair of clauses  $C_1$  and  $C_2$ 
3          if no new pair exists
4              then return FAIL
5           $R \leftarrow \text{Resolvent}(C_1, C_2)$ 
6          if  $R = \{\}$ 
7              then return FALSE
8          else  $S \leftarrow S \cup \{R\}$ 

```

2.9 The resolution method picks two clauses and generates a resolvent till it generates the empty clause. If that happens it returns the value of the input. If it cannot pick two clauses it returns “fail”. The key to efficiency is making the right choice while choosing the

value that the method only returns the valuation “false” when it can derive the empty clause. If it cannot, it returns “fail” and not “true”. This is because it can only say that it has not been able to show the formula to be true, which does not mean that the formula is true. To take a simple example given the set $\{P, Q\}$ we cannot say anything about the truth value.

the key to using the resolution method is to apply it to formulas that are unsatisfiable. The task in propositional logic taken up is to test whether the consequent logically follows from the premises. Given a set of premises $\{\alpha_1, \alpha_2, \dots, \alpha_n\}$ and the desired goal β , we want to determine if the formula,

$$((\alpha_1 \wedge \alpha_2 \wedge \dots \wedge \alpha_n) \supset \beta)$$

is satisfiable. Since the resolution method can only test for unsatisfiability, we can try the negation of the above formula,

$$\neg((\alpha_1 \wedge \alpha_2 \wedge \dots \wedge \alpha_n) \supset \beta)$$

and convert this into *CNF*, the form that is required by the resolution method.

$$\begin{aligned} \neg((\alpha_1 \wedge \alpha_2 \wedge \dots \wedge \alpha_n) \supset \beta) &\equiv \neg(\neg(\alpha_1 \wedge \alpha_2 \wedge \dots \wedge \alpha_n) \vee \beta) \\ &\equiv ((\alpha_1 \wedge \alpha_2 \wedge \dots \wedge \alpha_n) \wedge \neg\beta) \\ &\equiv ((\alpha'_1 \alpha'_2 \wedge \dots \wedge \alpha'_n) \wedge \neg\beta') \end{aligned}$$

To generate the input for the resolution method we simply need to negate the goal and add it to the set of premises. We also need to convert each of the premises and the negated goal into the clause (*CNF*) form denoted by $\{C_1, C_2, \dots, C_n\}$. One of the transformations is

done by resolution method is a *proof by contradiction*. We start with the set of premises, *negate* the consequent of the goal, and show that it *leads to a contradiction* (something that is false or not possible).

We will illustrate the method with the example we had seen earlier. We need to show that following is true.

$$(((P \wedge Q) \wedge (P \supset R) \wedge ((R \wedge S) \supset T) \wedge (\neg Q \vee S) \wedge (\neg U \wedge \neg V)) \supset T)$$

10.	R	from 1, 3
11.	$\neg S$	from 9, 10
12.	$\neg Q$	from 11, 5
13.	\square	from 2, 12

Resolution proof is often better visualized as a directed graph. Each node in the graph represents a clause. Nodes representing the premises have no incoming arcs. Every other node has two incoming arcs, from the clauses that it is resolved from. A node may have multiple outgoing arcs, since it may be used in any number of subsequent steps. Figure 12.10 below depicts the proof given above as a directed graph.

A resolution proof of a formula is not unique. There may be many different ways of deriving the contradiction. Figure 12.11 below shows another proof.

This new proof is in fact quite similar to the earlier direct proof by forward reasoning. In fact, the part of the proof that produces the clause T , and that is resolved with the negated goal clause $\neg T$ to produce S . One can also identify the step resolving Q and $(\neg Q \vee S)$ as the *disjunctive syllogism* rule producing S . Since one may substitute $(Q \supset S)$ for $(\neg Q \vee S)$ we can also see this as an application of the *modus ponens* rule. Resolution in fact subsumes the other rules, and as we will see when we look at first order resolution, it is a complete logic machine, whereas the other rules by themselves may not be complete. The problem of choosing the clauses still remains though, and a number of strategies have been proposed. We will look at a few of them here, and one more in the first order case.

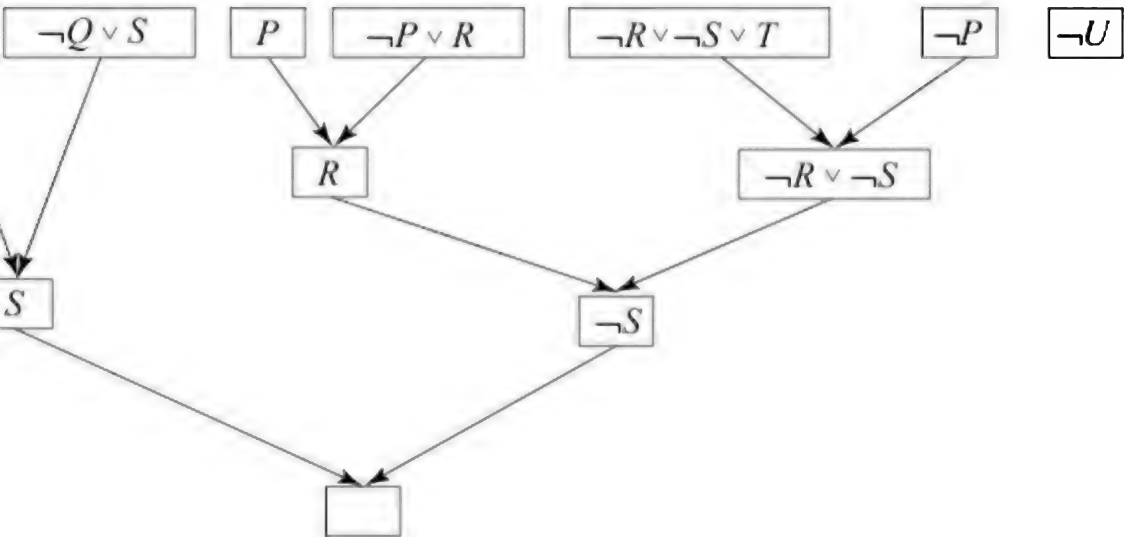


Figure 12.10 The resolution proof as a directed graph.

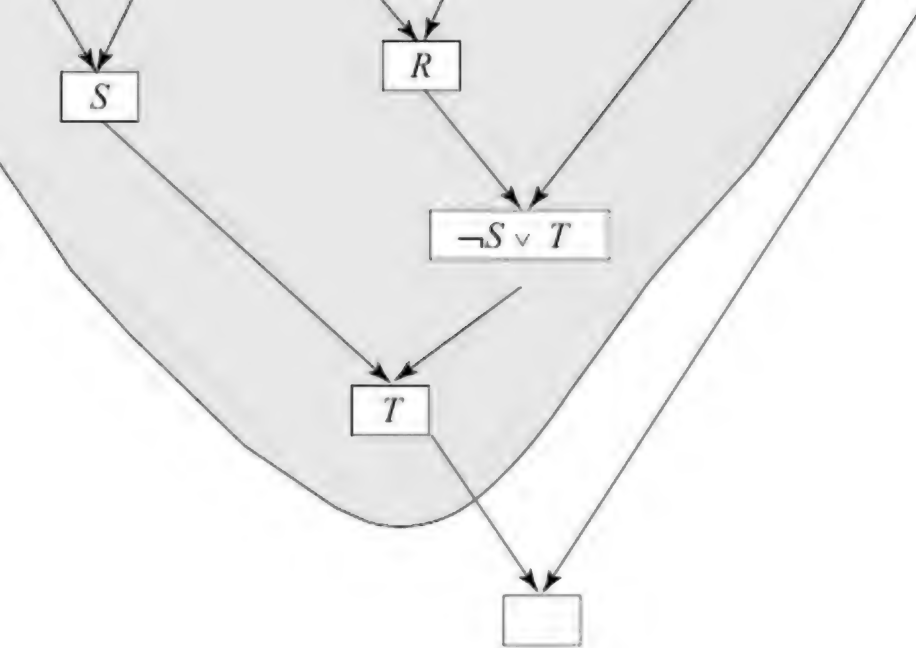


Figure 11 An alternative resolution proof.

Support

A clause represents a contradiction. The premises are presumed to be true. The contradiction arises from the negation of the goal to the set of clauses. The set of support strategy says that at every resolution step, the clauses must be the goal clause or derived from the goal clause. Neither of the two refutations shown in Figure 11 uses this strategy, but it is possible to find a refutation that does.

Unit Preference

The goal is to derive the null clause. The null clause has zero literals. Let the clauses being resolved have N and M literals. Then the resolvent will have $(N + M - 2)$ literals. This will be smaller than both parent clauses only if $N + M - 2 < N$ and $N + M - 2 < M$. The unit preference strategy says that one must prefer clauses of size 1. Then one can hope to reduce the size of the resolvent till it reaches size 0 for the null clause. Both examples above have used the unit preference strategy.

First Order Logic

The automatic argument we have looked at earlier is not possible in propositional logic. We could try and formalize the argument as follows.

P = Socrates is a man
 Q = Socrates is mortal

The goal would be Q , but to relate it to P we would need a rule of the form $(P \supset Q)$ which would read “If Socrates is a man then Socrates is mortal”. However what we have is “All men are mortal” which could also be written as “If someone is a man then that someone is mortal”. The mortality rule we have applied to “someone” but we are not inferring about Socrates.

To be able to identify Socrates with “someone” in the rule we have, we need to break up our sentences into their constituents, where we can talk about the individuals and their properties. We can do this by modifying the notation to use *predicates*. For the example in question we could now encode our sentences using the predicates “Man” and “Mortal” as follows.

FOL Syntax

Language of *FOL* is also defined in two parts, the logical part and the non-logical part.

In the propositional logic the logical part of the vocabulary contains symbols like “ \wedge ”, “ \vee ”, “ \neg ”, and “ \supset ”, sets of brackets, and the constant symbols “ \perp ” and “ \top ”. The language of *FOL* also uses a set of variables $V = \{v_1, v_2, v_3, \dots\}$ though we commonly use $\{x, y, z, x_1, y_1, z_1, \dots\}$ as well. In addition there are logical symbols “ \forall ” read as “for all”, and “ \exists ” read as “there exists”. The former is the *universal quantifier* and the latter is the *existential quantifier*²⁵. They are used to quantify the values a variable can take. Finally one may include the symbol “=” as “equals”.

The non-logical part of *FOL* constitutes of three sets.

1. Set of predicate symbols $\mathbb{P} = \{P_1, P_2, P_3, \dots\}$. We also use the symbols $\{P, Q, R, \dots\}$. More commonly we use words like “Man”, “Mortal”, “GreaterThan”. Each symbol has an arity associated with it, that stands for the number of arguments the predicate takes.

2. Set of function symbols $\mathcal{F} = \{f_1, f_2, f_3, \dots\}$. We commonly use the symbols $\{f, g, h, \dots\}$ or words like “Sum”, “Product”. Each function symbol has an arity that denotes the number of argument it takes.

3. Set of constant symbols $\mathbb{C} = \{c_1, c_2, c_3, \dots\}$. We often use symbols like “0”, or “Socrates”, or “Darjeeling”, which are meaningful to us.

These three sets define a specific language $L(P, F, C)$.

The basic constituents of *FOL* expressions are *terms*. The set of terms \mathfrak{T} of $L(P, F, C)$ is defined as follows. The constants and the variables are terms by definition. More terms are defined using the function symbols.

If $t \in V$ then $t \in \mathfrak{T}$

If $t \in \mathbb{C}$ then $t \in \mathfrak{T}$

If $t_1, t_2, \dots, t_n \in \mathfrak{T}$ and $f \in \mathcal{F}$ is an n -place function symbol then $f(t_1, t_2, \dots, t_n) \in \mathfrak{T}$

The set of formulas is defined using terms and predicate symbols. By default the logical symbols “ \perp ” and “ \top ” are formulas. The set of well formed formulas F of $L(P, F, C)$ is defined as follows.

Atomic formulas

$\perp \in F$

$\top \in F$

If $t_1, t_2 \in \mathfrak{T}$ then $(t_1 = t_2) \in F$

If $t_1, t_2, \dots, t_n \in \mathfrak{T}$ and $P \in \mathbb{P}$ is an n -place predicate symbol then $P(t_1, t_2, \dots, t_n) \in F$ Formulas²⁶.

If $\alpha \in F$ then $\neg \alpha \in F$

If $\alpha, \beta \in F$ then $(\alpha \wedge \beta) \in F$

If $\alpha, \beta \in F$ then $(\alpha \vee \beta) \in F$

If $\alpha, \beta \in F$ then $(\alpha \supset \beta) \in F$

In *FOL* the quantifiers are used to define formulas as well. A quantifier quantifies a variable that occurs in the formula, which is the formula immediately following it.

If $\alpha \in F$ and $x \in V$ then $\forall x (\alpha) \in F$

If $\alpha \in F$ and $x \in V$ then $\exists x (\alpha) \in F$

Having defined the set of formulas of *FOL* we can now define the set of sentences S_{FOL} .

The set of sentences S_{FOL} in a language $L(P, F, C)$ are all formulas without free variables. A variable in a formula is free if it is not in the scope of any quantifier.

FOL Semantics

The smallest unit in propositional logic is the sentence. The sentence can stand for something, and the sentence can be assigned a truth value. In *FOL* a sentence is made up of different kinds of symbols and the meaning of the sentence is to be determined by what the constituent parts stand for. Unlike propositional logic where a sentence stands for itself, in first order logic a sentence talks about elements of some domain. Terms of the language

then $t \in \mathfrak{I}$ mapped to the element of the domain D as follows.

If $t \in V$ then $t^A = t^A$

If $t \in \mathfrak{C}$ then $t^A = t^I$

If $t = f(t_1, t_2, \dots, t_n)$ then $t^A = f(t_1^A, t_2^A, \dots, t_n^A)$

credit for our notion of the predicate, its meaning and its truth value must go far back to Plato's

Plato associated the notion of forms with properties such as tall, white and the notion of sitting like "Theaetetus is sitting" the constituents "Theaetetus" and "is sitting" may be meaningful in the either true nor false. Their combination is both meaningful and capable of having a truth value.

terms stand for objects in the domain, and predicates stand for relations on the domain. The interpretation of formulas is as follows.

$\text{Val}(\top) = \text{true}$

$\text{Val}(\perp) = \text{false}$

$\text{Val}(t_1 = t_2)^A = \text{true}$ iff $t_1^A = t_2^A$

A predicate with variables is assigned a value *true* under some assignment function A if the corresponding tuple to the corresponding relation.

$\text{Val}(P(t_1, t_2, \dots, t_n))^A = \text{true}$ iff $\langle t_1^A, t_2^A, \dots, t_n^A \rangle \in P^I$

A sentence of $L(P, F, C)$ does not have any free variables. A sentence of the form $\exists x(\alpha)$ is true if there is some x for which the formula is true. For a sentence in which the variable is universally quantified then the sentence is true for all possible values of x . Formally,

$\text{Val}(\exists x(\alpha))^A = \text{true}$ iff α^B is true for some assignment B that is an x -variant²⁸ of A . In other words the formula α must be true for some value of x .

$\text{Val}(\forall x(\alpha))^A = \text{true}$ iff α^B is true for all possible assignments B that are x -variants of A . In other words the variable x must be true for any value and α must be true.

Compound formulas constructed by the use of logical connectives the truth values are arrived at in the same way as in propositional logic.

The meaning of the terms and sentences of a set of *FOL* sentences is given by an *interpretation* $\mathfrak{I} = \langle D, I \rangle$ where D is a domain and I is an interpretation mapping. An interpretation $M = \langle D, I \rangle$ of a set of sentences or a language $L(P, F, C)$ is a *model* if all the sentences in the set are *true* in the interpretation.

A sentence of *FOL* may be a propositional sentence when it does not have any variable. For example, "Juliet loves Romeo" (Loves(Romeo, Juliet)). Such sentences are often known as *facts*.

In addition there may be "facts" that are expressed using quantifiers. For example on the domain of (non-zero) numbers the following are always true²⁹.

$\exists x \text{ Even}(x)$

$\forall x \text{ Greater Than}(\text{Successor}(x), x)$

As we can observe that each quantified sentence is a short form for a propositional sentence, which may appear to be infinitely long. The sentences are,

$\text{Even}(0) \vee \text{Even}(1) \vee \text{Even}(2) \vee \dots$

$\text{Greater Than}(\text{Successor}(0), 0) \wedge \text{Greater Than}(\text{Successor}(1), 1) \wedge \dots$

As we can see that not only do quantifiers allow us to express things succinctly, they also allow us to express things that would not be possible in propositional logic. The universal quantifier combines propositional sentences using the connective \wedge , and the existential quantifier using \vee . They also allow us to describe relations between sets of propositional sentences. For example,

$\forall x(\text{Even}(x) \supset \neg \text{Odd}(x))$

$\forall x(\text{Man}(x) \supset \text{Mortal}(x))$

We also sometimes refer to such statements as *rules*, and one talks of a knowledge base containing such

$= \{\text{Next, Combine}\}$

$C = \{\text{begin}\}$

if one were dealing with the set of natural numbers, one would have the following interpretation.

ger(x, y):	x is greater than y
ual (x, y):	x is equal to y
xt(x):	stands for successor of x
mbine(x, y):	stands for $x+y$
jin:	stands for 0

the other hand one could be talking about strings over a particular alphabet. Then,

ger(x, y):	x is longer than y
ual (x, y):	x is same length as y
xt(x):	stands for successor of x in the lexicographic order
mbine(x, y):	stands for x concatenated with y
jin:	stands for the empty string

formulas and sentences in the given language would *mean* different things depending upon the domain of our own minds such symbol processing machines? How could they be purely syntactic machines without meaning about when we are so situated in our real world? Or are our real worlds a creation of our minds? We must believe? However we do (seem to) have our physical selves and we do live in our physical environment about it and acting on it. Perhaps it is the marvel of evolution that our perceptive system has evolved to process a processing mind (Dawkins, 1996).

FOLRules

propositional logic rules we saw earlier are valid in *FOL* as well. In addition we need new rules to handle quantified statements. The two commonly used rules of inference are,

$\frac{\forall x P(x)}{P(a)}$	where $a \in \mathbb{C}$	Universal Instantiation
-------------------------------	--------------------------	-------------------------

$\frac{P(a)}{\exists x P(x)}$	where $a \in \mathbb{C}$	Generalization
-------------------------------	--------------------------	----------------

Following rules of substitution are also useful,

$\neg \forall x \alpha$	\equiv	$\exists x \neg \alpha$	De Morgan's law
$\neg \exists x \alpha$	\equiv	$\forall x \neg \alpha$	De Morgan's law
$\forall x \forall y \alpha$	\equiv	$\forall y \forall x \alpha$	
$\exists x \exists y \alpha$	\equiv	$\exists y \exists x \alpha$	

The following rules the notation $A[x]$ stands for the fact that A is a formula containing x , while A stands for a formula that does not contain x (as in (Manna, 1974). Likewise for B .

$\exists x A[x] \vee B$	\equiv	$\exists x (A[x] \vee B)$
$A \vee \exists x B[x]$	\equiv	$\exists x (A \vee B[x])$
$\forall x A[x] \vee B$	\equiv	$\forall x (A[x] \vee B)$
$A \vee \forall x B[x]$	\equiv	$\forall x (A \vee B[x])$
$\forall x A[x] \wedge \forall x B[x]$	\equiv	$\forall x (A[x] \wedge B[x])$
$\forall x A[x] \wedge B$	\equiv	$\forall x (A[x] \wedge B)$
$A \wedge \forall x B[x]$	\equiv	$\forall x (A \wedge B[x])$
$\exists x A[x] \wedge B$	\equiv	$\exists x (A[x] \wedge B)$
$A \wedge \exists x B[x]$	\equiv	$\exists x (A \wedge B[x])$
$\forall x A[x] \supset \exists x B[x]$	\equiv	$\exists x (A[x] \supset B[x])$
$\exists x (A[x] \supset B)$	\equiv	$\forall x (A[x] \supset B)$
$B \supset \exists x A[x]$	\equiv	$\exists x (B \supset A[x])$
$\forall x A[x] \supset B$	\equiv	$\exists x (A[x] \supset B)$
$B \supset \forall x A[x]$	\equiv	$\forall x (B \supset A[x])$

Some of these may be required to convert a first order formula into clause form, a form suitable for the resolution algorithm. The reader is encouraged to prove the above equivalences. Many of the seemingly strange equivalences around the operator \supset can be explained by the fact that $(\alpha \supset \beta)$ has a hidden negation sign inside, making it the equivalent form $(\neg\alpha \vee \beta)$. When the negation moves across a quantifier it transforms the existential into a universal and vice versa.

Forward Chaining in FOL

Rephrase our example (Alice) problem in first order terminology.

Alice likes mathematics and she likes stories.

If someone likes mathematics she likes algebra³⁰.

If someone likes algebra and likes physics she will go to college.

Alice does not like stories or she likes physics.

Alice does not like chemistry and history.

We can formalize the statements in FOL as follows.

$\text{likes}(\text{Alice}, \text{Math}) \wedge \text{likes}(\text{Alice}, \text{stories})$

$\forall x (\text{likes}(x, \text{Math}) \supset \text{likes}(x, \text{Algebra}))$

$\forall x ((\text{likes}(x, \text{Algebra}) \wedge \text{likes}(x, \text{Physics})) \supset \text{goesTo}(x, \text{College}))$

$\text{likes}(\text{Alice}, \text{stories}) \vee \text{likes}(\text{Alice}, \text{Physics})$

$\text{likes}(\text{Alice}, \text{Chemistry}) \wedge \neg \text{likes}(\text{Alice}, \text{History})$

We can now generate a proof that is analogous to the proof in propositional logic.

$\text{likes}(\text{Alice}, \text{Math})$

can compare this proof with one in the propositional logic. Except for the two steps using substitution (UI) the other steps are identical. And it is these UI steps that make it difficult to automate the process. Of all the constants in the domain, and one imagines there must be many in a realistic problem, which rule should the rule select? In the example above the variable x is *bound* to Alice somehow. Of course this is because it is about Alice that we want to make the inference from liking math to liking algebra. To bound x to Alice our inference engine (proof finding procedure) will need to search the set of statements to find the relevant constants to instantiate to. Fortunately there exists a representation, the *implicit quantifier*, that makes this a lot easier. Before we look at the implicit quantifier form, let us digress a little and look at a simple example. While applying UI to the rule “ $\forall x(\text{likes}(x, \text{Math}) \supset \text{likes}(x, \text{Algebra}))$ ” what stops us from binding x to ‘Romeo’ which is also a constant term? One approach way would be to organize the constants into *categories*. In our example we could introduce the categories “person”, “subject”, and “educationInstitute”. Let us use the categories “boy” and “girl”. Then we need to add more data about the categories of the constant symbols: person(Romeo), girl(Juliet), subject(Physics), subject(Math), and so on. Let us look at some statements and how they are formalized.

“If someone likes maths she likes algebra.”
 $\forall x((\text{person}(x) \wedge \text{likes}(x, \text{Math}) \supset \text{likes}(x, \text{Algebra}))$

Knowing that we bind variables starting from the left, then x could only be bound to something that belongs to the “person” category. And constants like “Physics” will be ruled out. Here are some more statements,

“Every boy likes a girl.”
 $\forall x (\text{boy}(x) \supset \exists y (\text{girl}(y) \wedge \text{likes}(x, y))$

Is this different from “Every boy *loves* a girl.”? Do we need to introduce a predicate for every word in the language? Then if we wanted to assert that “if someone loves/likes someone then they care for them” we would have separate rules for love and like? Could the above sentence be formalized as the following?

$\exists y (\text{girl}(y) \wedge \forall x (\text{boy}(x) \supset \text{likes}(x, y))$

The English language is rich and ambiguous enough to allow both meanings. One of the advantages of formalization is that one gets rid of such ambiguity. In the following we can use the same predicate “likes” to allow both meanings.

“Every boy likes a course.”
 $\forall x (\text{boy}(x) \supset \exists y (\text{course}(y) \wedge \text{likes}(x, y))$

If we introduce categories like “boy” and “person” do we need to put in data for both? Or can we use the same rules like,

$\forall x (\text{boy}(x) \supset \text{person}(x))$

We will explore some of these knowledge representation issues in the next chapter. Meanwhile let us return to automated reasoning.

Skolemization

Formulas that contain both universally quantified and existentially quantified variables. However, we will use rules that have only universally quantified variables. Rules of inference typically say “If there is an x such that $P(x)$ is true then $Q(x)$ holds too”. Though x sounds existential in the above reading – “there is an x such that $P(x)$ is true” – it is a universally quantified variable. The equivalent statement is “For all x such that $P(x)$...”. We will express these rules in terms of ground statements.

$\{\text{antecedents}\} \supset \text{consequent}$

Forward chaining picks a rule, whose antecedents have matching facts, and produces matching consequents. If all the antecedents are ground instances then this process involves creating a matching ground instance of the consequent and then applying modus ponens to that ground instance³¹. The variables involved are universally quantified. The process continues until a goal is reached below.

In the following predicate symbols, logical operators, function symbols are all treated as constants, in the sense that they must match exactly. A variable, on the other hand, is one that can be substituted by anything else (a term). We assume a function *length* is available to determine the length of a list. Two lists are unified if they have the same length. The following algorithm is along the lines of the version presented by Robinson (1965) and McDermott, (1985).

```

unify(list1, list2)
  return SubUnify(list1, list2, ())

SubUnify(list1, list2, theta)
  if "fail" ∈ theta
    then return FAIL
  if Var(list1)
    then return VarUnify(list1, list2, theta)
  if Var(list2)
    then return VarUnify(list2, list1, theta)
  if Constant(list1)
    then if list1 = list2
          then return theta
          else return FAIL
  if Constant(list2)
    then return FAIL
  if list1 = list2 = ( )
    then return theta
  if Length(list1) ≠ Length(list2)
    then return FAIL
  else return Append( SubUnify(Head(list1), Head(list2), theta),
                        SubUnify(Rest(list1), Rest(list2), theta)

VarUnify(variable, list, theta)
  if ExistsIn(variable, list)
    then return FAIL
  if (variable value) ∈ theta /* the <variable, value> pair is in theta
    then return SubUnify(list, value, theta)
    else return Cons((variable, list), theta)

```

13 The unification algorithm compares the two inputs (lists) element by element making recursive calls where necessary. The function *ExistsIn* returns the substitution *theta* if it is consistent to do so. The function *ExistsIn* looks for occurrence of the variable in the list that it is. It can be implemented by first flattening the list and then checking for membership.

If the same variable name is repeated in two formulas then it could lead to a problem in the algorithm. Consider the following formulas,

(SmallerOrEqualThan 0 ?z)
 (SmallerOrEqualThan ?z, (successor ?z))

The first formula says that 0 is smaller than or equal to anything. The second says that any number is smaller than its successor.

If we run our unification algorithm on the two formulas the following will happen.

Since both inputs are lists of length three, three recursive calls to sub-unify will be made. In the first call the two constants match and it returns the empty substitution.

algorithm proceeds as before for the first two recursive call. The third call now is different -var or ?z), ((?z 0)). This time the algorithm returns the unifier theta = ((?x (successor ?z) (?z 0)). App on correctly unifies the two formula into (SmallerOrEqualThan 0 (successor 0)).

Handling Existential Quantifiers

said that the implicit quantifier form of a formula implicitly defines variables to be universally quantified. What if a first order formula has existentially quantified variables? The skolemization process replaces existentially quantified variables either by special constants known as Skolem constants or by special functions known as Skolem functions.

If an existential quantifier is not in the scope of any universal quantifier, then the variable it quantifies is replaced by a Skolem constant. For example, the statements,

$$\begin{aligned} &\exists z (\text{Student}(z) \wedge \text{Bright}(z)) \\ &\exists y (\text{girl}(y) \wedge \forall x (\text{boy}(x) \supset \text{likes}(x, y)) \end{aligned}$$

are skolemized as,

$$\begin{aligned} &((\text{Student } \text{sk1}) \wedge (\text{Bright } \text{sk1})) \\ &((\text{girl } \text{sk2}) \wedge ((\text{boy } ?x) \supset (\text{likes } x, \text{sk2}))) \end{aligned}$$

A Skolem constant is not a real constant in the sense that we do not know what it maps to in the domain. It may map to some element. It may also map to more than one element, as for example in the first statement there may be more than one bright student³².

If an existential quantifier is in the scope of one or more universal quantifiers then the existentially quantified variable is replaced by a Skolem function of the corresponding universally quantified variables. For example the statement

$$\begin{aligned} &\forall x \forall y \exists z (\text{LessThan}(x, z) \wedge \text{LessThan}(y, z)) \\ &\forall x (\text{boy}(x) \supset \exists y (\text{girl}(y) \wedge \text{likes}(x, y)))^{33} \end{aligned}$$

are skolemized as,

$$\begin{aligned} &((\text{LessThan } ?x (\text{sk57 } ?x ?y)) \wedge (\text{LessThan } ?y (\text{sk57 } ?x ?y))) \\ &((\text{boy } ?x) \supset ((\text{girl } (\text{sk16 } ?x)) \wedge (\text{likes } x (\text{sk16 } ?x)))) \end{aligned}$$

sk57 and sk16 are Skolem functions. A Skolem function is not a mapping from the arguments to the value. Rather it says that the value is dependent upon the arguments in some way. The expression (sk57 ?x ?y) is some number that is greater than x and y. Likewise (sk16 ?x) is some girl who x likes. While convenient, we may use names of Skolem functions meaningful to us, like (greaterThanBoth ?x ?y) or (LikedBy ?x). Whether a variable is universally quantified or existentially quantified has to be decided carefully. One must be careful that a negation sign influences the nature of the quantifier. Consider the formalization of “An immortal exists” which is another way of saying that all men are mortal.

$$\exists x (\text{Man}(x) \wedge \neg \text{Mortal}(x))$$

What is the nature of the variable x? On the surface it is bound by an existential quantifier so one might formalize it as $\neg((\text{Man } \text{sk11}) \wedge \neg(\text{Mortal } \text{sk11}))$ but that only talks of a specific, albeit unspecified, individual. The correct way to skolemize a formula is to first push the negation sign inside. That gives us the

$$\forall x \neg(\text{Man}(x) \wedge \neg \text{Mortal}(x))$$

which reveals its true form, being a universally quantified variable. The following somewhat disconcerting example illustrates the fact that the antecedent in an implication statement also has negation lurking inside. It reads “If there exists a number that is even and odd then the Earth is flat” and is formalized as,

$$(\exists x (\text{Number}(x) \wedge \text{Even}(x) \wedge \text{Odd}(x))) \supset \text{Flat}(\text{Earth}))$$

However if we rewrite the formulas as,

$$\neg(\exists x (\text{Number}(x) \wedge \text{Even}(x) \wedge \text{Odd}(x))) \vee \text{Flat}(\text{Earth}))$$

$$\begin{aligned} &\equiv \forall x \forall y (\neg \text{Detective}(x) \vee \neg \text{Sidekick}(y, x) \vee \text{Successful}(y, x)) \\ &\equiv \forall x \forall y (\neg (\text{Detective}(x) \wedge \text{Sidekick}(y, x)) \vee \text{Successful}(y, x)) \\ &\equiv \forall x \forall y (\neg (\text{Detective}(x) \wedge \text{Sidekick}(y, x)) \supset \text{Successful}(y, x)) \end{aligned}$$

In the unification algorithm the Skolem constants and function names are simply treated as constants. We'll illustrate this with a couple of examples.

$$\begin{array}{ll} \text{Premise 1:} & \exists x \text{ Even}(x) \\ \text{Premise 2:} & \forall x (\text{Even}(x) \supset \neg \text{Odd}(x)) \\ \text{Premise 3:} & \exists x \neg \text{Odd}(x) \end{array}$$

If we skolemize the premises, we get,

$$\begin{aligned} &(\text{Even } \text{SomeEvenNumber}) \\ &((\text{Even } ?x) \supset \neg \text{Odd}(x)) \end{aligned}$$

Using the substitution $\{?x = \text{SomeEvenNumber}\}$ we can infer $\neg \text{Odd}(\text{SomeEvenNumber})$. The second example involves, "Everyone loves someone" and "If someone loves somebody then they care for them". The goal is to prove that everyone cares for someone. Assuming that the universe of discourse is people we can formalize the premises as follows:

$$\begin{array}{ll} \text{Premise 1:} & \forall x \exists y \text{ Loves}(x, y) \\ \text{Premise 2:} & \forall x \forall y (\text{Loves}(x, y) \supset \text{CaresFor}(x, y)) \\ \text{Premise 3:} & \forall x \exists y \text{ CaresFor}(x, y) \end{array}$$

If we skolemize the premises we get,

$$\begin{aligned} &(\text{Loves } ?x (\text{sk7 } ?x)) \\ &(\text{Loves } ?z ?y \supset (\text{CaresFor } ?z ?y)) \end{aligned}$$

Using the substitution $\{?z = ?x, ?y = (\text{sk7 } ?x)\}$, we get the conclusion,

$$(\text{CaresFor } ?x (\text{sk7 } ?x))$$

Completeness of Forward Chaining

Forward chaining is a form of reasoning that is used to unravel the information that is implicit in a set of sentences. Forward chaining, also known as data-driven reasoning, has a problem with handling disjunctive facts. This is because forward chaining often involves reasoning by cases.

The technique that allows reasoning with disjunctive facts is the constructive dilemma that says,

$$\begin{array}{ll} \text{From} & (\alpha \supset \beta) \wedge (\gamma \supset \delta) \\ \text{and} & \alpha \vee \gamma \\ \hline \text{Infer} & \beta \vee \delta \end{array}$$

We can employ the rule to solve the following problem,

$$\begin{array}{ll} \text{Premise 1:} & \forall x (\text{Winner}(x) \supset \text{PhotoInPaper}(x)) \\ \text{Premise 2:} & \text{Winner}(\text{Sunil}) \vee \text{Winner}(\text{Anil}) \\ \text{Premise 3:} & \text{PhotoInPaper}(\text{Sunil}) \vee \text{PhotoInPaper}(\text{Anil}) \end{array}$$

Without the absence of appropriate rules of inference it cannot extract some kinds of implicit information. We'll illustrate this with two examples. The first appears in (Charniak and McDermott, 1985) and the second one in (Brachman and Levesque, 1985).

The first problem says that given the facts that block A and block B are on the table, and that at least one of the blocks is green, it is true that there exists a green block on the table. The premises are expressed together as follows:

Goal is indeed true, because if B is green then A on B satisfies the query. Else B is not green and B is not green. Again forward chaining is unable to handle this problem.

Observe that both the problems involve an existential goal. Such goals – “is there an $X \dots$?” are quite common. The difficulty however arises because of the presence of disjunctive facts. In the second problem we fully add fact $(\text{Green}(B) \vee \neg \text{Green}(B))$. Reasoning with disjunctive facts often involves reasoning with one develops two or more strands or chains of inferences and show that they combine to a consistent result. The forward chaining procedure on the other cannot reason by cases.

In principle of course one can add more rules on the lines of the constructive dilemma rule, but that is messy, ad hoc, and too specific. For example, one could add the following rule to solve the first problem.

From: β
 and: δ
 and: $\frac{\alpha \vee \gamma}{}.$
 Infer $(\beta \wedge \alpha) \vee (\delta \wedge \gamma)$

One might think that would be a specific rule for that particular inference. Observe that both the problems can easily be solved by a proof by contradiction. Assume the negation of the goal, and show that it leads to a contradiction. This is what Robinson's *resolution method* does. Further the resolution method uses only one rule of inference and is complete.

To apply the resolution method we first need to convert the first order sentences into the clause form, which is a normal form of the CNF in FOL.

Clause Form

A first order formula is in clause form if it is of the following form,

$$\forall x_1 \forall x_2 \dots \forall x_N (C_1 \wedge C_2 \dots \wedge C_N)$$

Each C_i is a *clause* made of disjunction of *literals*, and each literal is an atomic formula or its negation. The clause form contains only universal quantifiers. The consequence of having only universal quantifiers bunched up in the left is that one can in fact ignore the quantifiers during processing. That makes the clause form a little bit simpler.

It is shown by Thoralf Skolem that every formula can be converted into the clause form. The procedure for converting a first order formula into the clause form is as follows (see also (Manna, 1974), (Brachman and Levesque, 1995)). Given a FOL formula a ,

1. Take the existential closure of a . This ensures that there are no free variables in the formula.
2. Standardize variables apart across quantifiers. Rename variables so that the same symbol does not denote different quantifiers.
3. Eliminate all occurrences of operators other than \wedge , \vee , and \neg .
4. Move \neg all the way in.
5. Push the quantifiers to the right. This ensures that their scope is as tight as possible.
6. Eliminate \exists .
7. Move all \forall to the left. They can be ignored henceforth.
8. Distribute \wedge over \vee .
9. Simplify.
10. Rename variables in each clause (disjunction).

At the end of this procedure the given first order formula is converted into an equivalent clause form. One will need only a few of the above steps. As an example let us convert the following using the steps 1-10.

6. $((\text{girl}(\text{sk-}y) \wedge \forall x(\neg \text{boy}(x) \vee \text{likes}(x, y)) \vee (\text{boy}(\text{sk-}x_1) \wedge \text{girl}(\text{sk-}z) \wedge \text{loves}(\text{sk-}x_1, \text{sk-}z))) \wedge \neg \text{loves}(\text{sk-}x_1, \text{sk-}z)) \vee (\text{boy}(\text{sk-}x_1) \wedge \text{girl}(\text{sk-}z) \wedge \text{loves}(\text{sk-}x_1, \text{sk-}z) \wedge \neg \text{loves}(\text{sk-}x_1, \text{sk-}z))$
7. $\forall x((\text{girl}(\text{sk-}y) \wedge (\neg \text{boy}(x) \vee \text{likes}(x, y)) \vee (\text{boy}(\text{sk-}x_1) \wedge \text{girl}(\text{sk-}z) \wedge \text{loves}(\text{sk-}x_1, \text{sk-}z) \wedge \neg \text{loves}(\text{sk-}x_1, \text{sk-}z))) \wedge \neg \text{loves}(\text{sk-}x_1, \text{sk-}z))$
8. $(\text{girl}(\text{sk-}y) \vee \neg \text{boy}(x) \vee \text{boy}(\text{sk-}x_1)) \wedge (\text{girl}(\text{sk-}y) \vee \text{likes}(x, y) \vee \text{boy}(\text{sk-}x_1)) \wedge (\text{girl}(\text{sk-}y) \vee \neg \text{boy}(x) \vee \text{girl}(\text{sk-}z)) \wedge (\text{girl}(\text{sk-}y) \vee \text{likes}(x, y) \vee \text{girl}(\text{sk-}z)) \wedge (\text{girl}(\text{sk-}y) \vee \neg \text{boy}(x) \vee \text{loves}(\text{sk-}x_1, \text{sk-}z)) \wedge (\text{girl}(\text{sk-}y) \vee \text{likes}(x, y) \vee \text{loves}(\text{sk-}x_1, \text{sk-}z))$
9. no change
10. $(\text{girl}(\text{sk-}y) \vee \neg \text{boy}(x_A) \vee \text{boy}(\text{sk-}x_1)) \wedge (\text{girl}(\text{sk-}y) \vee \text{likes}(x_B, y_B) \vee \text{boy}(\text{sk-}x_1)) \wedge (\text{girl}(\text{sk-}y) \vee \neg \text{boy}(x_C) \vee \text{girl}(\text{sk-}z)) \wedge (\text{girl}(\text{sk-}y) \vee \text{likes}(x_D, y_D) \vee \text{girl}(\text{sk-}z)) \wedge (\text{girl}(\text{sk-}y) \vee \neg \text{boy}(x_E) \vee \text{loves}(\text{sk-}x_1, \text{sk-}z)) \wedge (\text{girl}(\text{sk-}y) \vee \text{likes}(x_F, y_F) \vee \text{loves}(\text{sk-}x_1, \text{sk-}z))$

different avatars of variable x have been standardized apart (x_A, x_B, \dots, x_F). This does not change the meaning of the sentence, since each of them still represents a universally quantified variable. That is, each of them can take any value.

Resolution Refutation in FOL

With the propositional logic the procedure for finding a proof by the resolution refutation method is as follows:

1. Convert each premise into clause form
2. Negate the goal and convert it into clause form
3. Add the negated goal to the set of clauses
4. Choose two clauses such that two opposite signed literals in them can be unified
5. Resolve the two clauses using the MGU and add the resolvent to the set
6. Repeat steps 4–5 till a null resolvent is produced

For the sake of completeness the resolution rule is defined as follows. A literal is an atomic formula. A clause is a disjunction of literals. Let C_i and C_k be two clauses with the structure,

$$C_i = (L_1 \vee L_2 \vee \dots \vee L_k \vee P_1 \vee P_2 \vee \dots \vee P_n) \\ C_k = (\neg R_1 \vee \neg R_2 \vee \dots \vee \neg R_s \vee Q_1 \vee Q_2 \vee \dots \vee Q_t)$$

Let θ be the MGU for $\{L_1, L_2, \dots, L_k, R_1, R_2, \dots, R_s\}$ then we can resolve C_i and C_k to give us the resolvent,

$$(P_1\theta \vee P_2\theta \vee \dots \vee P_n\theta \vee Q_1\theta \vee Q_2\theta \vee \dots \vee Q_t\theta)$$

where we throw away all the positive literals L_j and negative literals $\neg R_i$, and combine the remainder after substitution θ . Let us see a few examples.

Three clauses for the Socratic argument are.

$= (\neg(\text{Man } ?x)) \vee (\text{Mortal } ?x)$	premise
$= (\text{Man Socrates})$	Premise
$= (\neg(\text{Mortal Socrates}))$	negated goal

Following resolvents are generated,

$= (\text{Mortal Socrates})$	$C_1, C_2, \{?x = \text{socrates}\}$
$= \perp$	C_3, R_1

Remember that applying a substitution $\{?x = \text{socrates}\}$ is a kind of instantiation. When the value is a constant, it is a restriction, but when the value is a function then it may simply be a restriction on the value the variable can take. A constant substitution is just an extreme case of restriction to one value.

Does showing something on a restricted value be enough to “prove” the generalized formula? The problem is that because the resolution refutation method is used to show that the input formula is unsatisfiable, it only works if the formula is a universally quantified formula. Let us say we are trying to show that $\forall x \alpha(x)$.

at the last clause come from the premises. The last clause is the negated goal clause. We show this in Figure 12.14.

We have chosen one of the many graphs that derive the null clause. This particular derivation actually proceeds by resolving the negated goal with the first clause, and then derives the contradiction using the negated goal. This might lead one to mistakenly believe that resolution is just another form of what forward chaining does. In fact it is not so. Resolution can do more than forward chaining can do. It is a complete method. That means that it can prove all true formulas that are derivable from the premises. To illustrate that, let us look at the green blocks problems and see how resolution can find a proof for them.

The first problem says that given,

$\text{Ontable}(A) \wedge \text{Ontable}(B) \wedge (\text{Green}(A) \vee \text{Green}(B))$

We need to show $\exists x (\text{Ontable}(x) \wedge \text{Green}(x))$. The negation of the goal is $\forall x (\neg \text{Ontable}(x) \vee \neg \text{Green}(x))$. The clauses are given below.

- | | |
|---|--------------|
| 1. $\text{Ontable}(A)$ | premise |
| 2. $\text{Ontable}(B)$ | premise |
| 3. $\text{Green}(A) \vee \text{Green}(B)$ | premise |
| 4. $\neg(\text{Ontable}(x)) \vee \neg(\text{Green}(x))$ | negated goal |

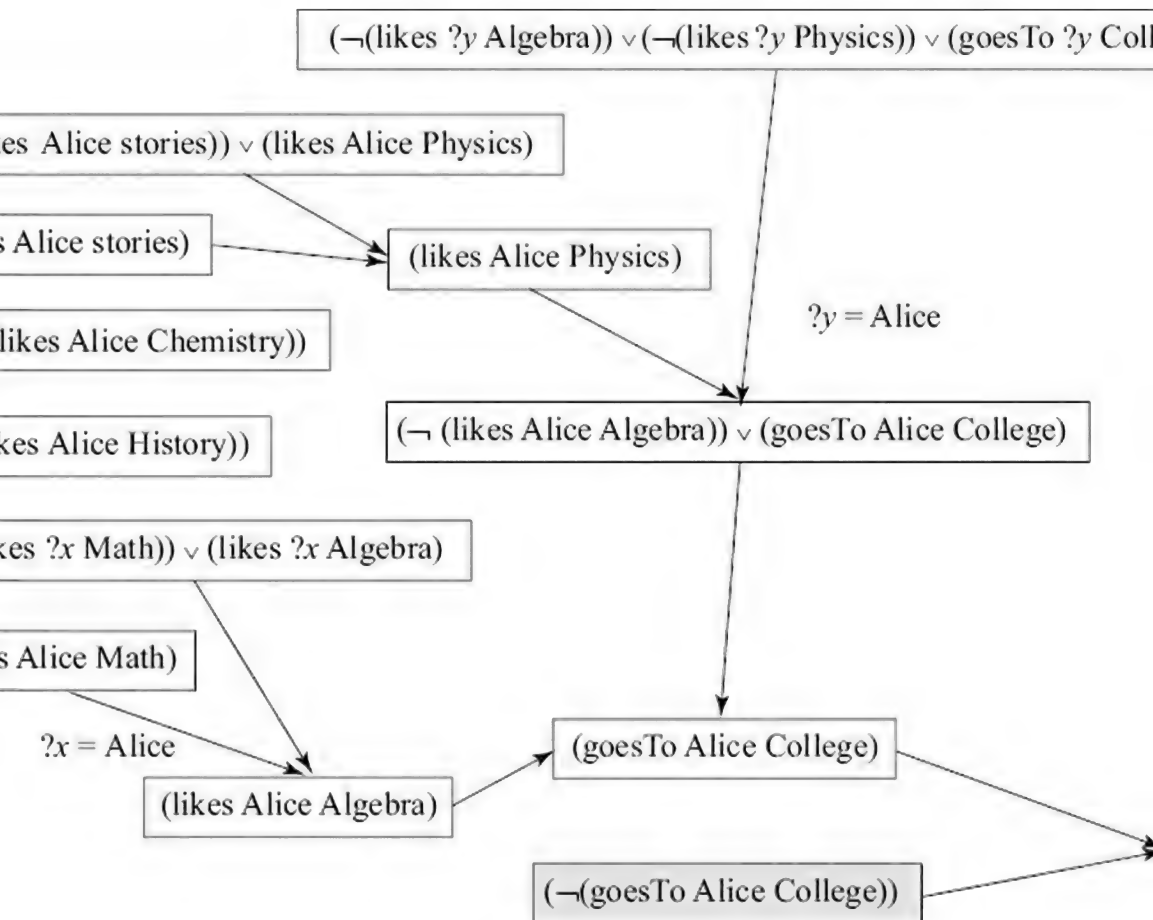


Figure 12.14 A resolution refutation directed graph for the Alice problem. The negated goal, the source of the contradiction is shown in the shaded box.

The possible derivation is,

- | | |
|------------------------------|----------------|
| 5. $\neg(\text{Green}(A))$ | 1, 4, $?x = A$ |
| 6. $\text{Green}(B)$ | 3, 5 |
| 7. $\neg(\text{Ontable}(B))$ | 1, 2, 6, 7 |

7. $(\neg (\text{On } A \text{ ?}y)) \vee (\neg (\text{Green ?}y))$	3,5, $?x = A$
8. $(\neg (\text{Green } B))$	1,7, $?y = B$
9. $(\text{Green } B)$	2, 6, $?x = A$
10. \perp	8, 9

These problems it did not matter but ideally one should rename a variable every time a new clause is added to avoid resolving two clauses with different variables having the same name.

Deductive Retrieval

Existential goals can also result in generating specific answers. Thus theorem proving can also be used to find answers to queries. The inference making ability enables the system to dig out facts that are implicit in the database. Theorem proving allows us to do more than just data base retrieval. We call this process as deductive retrieval. The simplest example comes from the database $\{\text{Man}(\text{Socrates}), \forall x(\text{Man}(x) \supset \text{Mortal}(x))\}$ and the query $\exists y(\text{Mortal}(y))$. The goal literally reads “there exists someone who is mortal”. It may also be read as a query “who is mortal?”. The substitution, or instantiation, used in the theorem proving process will yield the answer. At the process in the resolution refutation setting, the premises yield the following clauses,

$\text{Man}(\text{Socrates})$
 $(\text{Man ?}x) \vee (\text{Mortal ?}x)$

The negated goal is $\neg \exists y \text{Mortal}(y)$ which is the same as $\forall x \neg \text{Mortal}(y)$, which gives us the clause $(\neg (\text{Mortal ?}y))$. To extract the answer to the query we can add an *answer predicate* to the goal. We revise the goal to $(\neg (\text{Mortal ?}y)) \wedge \text{Answer}(y)$. We also need to modify the termination criteria from the null clause to include the presence of an answer clause. The resulting negated clause is,
 $(\neg (\text{Mortal ?}y)) \vee (\neg (\text{Answer ?}y))$

Since the *Answer* predicate will not play any role in the proof finding process we can discard the negation of the answer clause. We interpret it as the value of the answer variable.

$(\text{Mortal ?}y) \vee (\text{Answer ?}y)$

The same resolution method is used till we are left with an answer clause.

$\text{Mortal}(\text{Socrates})$ 1, 2, $?x = \text{Socrates}$
 $\text{Answer}(\text{Socrates})$ 3,4, $?y = \text{Socrates}$

Thus we have the answer as Socrates.

Let us try it with the problem of determining if there is a green block on the table we had solved earlier. There is no determinate answer. We have modified the goal clause in the following to include the answer predicate.

$(\text{Ontable } A)$	premise
$(\text{Ontable } B)$	premise
$(\text{Green } A) \vee (\text{Green } B)$	premise
$(\neg (\text{Ontable ?}x)) \vee (\neg (\text{Green ?}x)) \vee (\text{Answer ?}x)$	negated goal
$(\neg (\text{Green } A)) \vee (\text{Answer } A)$	1,4, $?x = A$
$(\text{Green } B) \vee (\text{Answer } A)$	3,5
$(\neg (\text{Ontable } B)) \vee (\text{Answer } B) \vee (\text{Answer } A)$	4, 6, $?x = A$
$(\text{Answer } B) \vee (\text{Answer } A)$	2,7

The algorithm correctly terminates with the answer that it is either block A or block B.

There are some situations however when resolution method may not extract the answer because it can find a refutative proof. Let us look at a small example to illustrate this.

Deduction is involved with unearthing hidden information in a set of facts. It is a valuable tool in the arsenal of a detective who has to solve a crime³⁵. Let us apply deduction to yet another fictitious crime mystery. The facts are as follows,

$\neg w \vee \neg x \vee \neg y \vee \neg z ((\text{Loc}(w, x, y) \wedge \neg(x = z)) \supset \neg(\text{Loc}(w, z, y))$

$\neg(\text{Mumbai} = \text{Chennai})$

$\neg(\text{Chennai} = \text{Mumbai})$

goal is $\exists x (\text{Culprit}(x))$ along with the answer predicate. We first convert the premises into clause notations as before, and then add the negated goal along with the answer predicate.

1. $(\text{Culprit Tinker}) \vee (\text{Culprit Tailor}) \vee (\text{Culprit Butler})$	premise
2. $(\neg (\text{Culprit } ?x_1)) \vee (\text{Tall } ?x_1)$	premise
3. $(\neg (\text{Culprit } ?x_2)) \vee (\text{Dark } x_2)$	premise
4. $(\neg (\text{Culprit } ?x_3)) \vee (\text{Loc } ?x_3 \text{ Chennai June 10})$	premise
5. $(\text{Loc Tailor Mumbai June 10})$	premise
6. (Short Tinker)	premise
7. $(\neg (\text{Short } x_4)) \vee (\neg (\text{Tall } ?x_4))$	premise
8. $(\neg(\text{Loc } ?w ?x ?y)) \vee (?x = ?z) \vee (\neg(\text{Loc } ?w ?z ?y))$	premise
9. $\neg(\text{Mumbai} = \text{Chennai})$	premise
10. $\neg(\text{Chennai} = \text{Mumbai})$	premise
11. $(\neg (\text{Culprit } ?c)) \vee (\text{Answer } ?c)$	negated

reader should verify that the given information is enough to *deduce* that the Butler is the culprit. However, the method may terminate with bindings that do not reveal anything. In fact, clauses 1 and 11 are equivalent to the null clause.

2. $\text{Culprit}(\text{Tailor}) \vee \text{Culprit}(\text{Butler}) \vee (\text{Answer Tinker})$	1, 11, $?c = \text{Tinker}$
3. $\text{Culprit}(\text{Butler}) \vee \text{Answer}(\text{Tailor}) \vee (\text{Answer Tinker})$	12, 11, $?c = \text{Tailor}$
4. $\text{Answer}(\text{Butler}) \vee \text{Answer}(\text{Tailor}) \vee (\text{Answer Tinker})$	13, 11, $?c = \text{Butler}$

What happens because statement 1 says that one of the three is the culprit while the negated goal says that no one is the culprit. These two statements in themselves are a contradiction. Instead of choosing the general goal, the reader is encouraged to try out specific goals like $\text{Culprit}(\text{Tinker})$ and $\text{Culprit}(\text{Butler})$ and verify that they indeed be proved.

Handling Equality

Look at a larger example that also deals with the equality statement. Let a database of families be defined with the mother and father relationships expressed as functions. The other relationships are defined in terms of the parent relationship. In the interest of brevity, we choose the following predicate and function symbols.

$M(X)$	function "mother of X "
$F(X)$	function "father of X "
$P(X, Y)$	predicate " X is parent of Y "
$S(X, Y)$	predicate " X is sibling of Y "
$G(X, Y)$	predicate " X is grandparent of Y "
$C(X, Y)$	predicate " X is cousin of Y "

The known data of the *Antararashtriya* family have the following parent relationships expressed in terms with equality.

$M(\text{Arushi}) = \text{Abigale}$
 $F(\text{Arushi}) = \text{Anandan}$
 $M(\text{Anna}) = \text{Abigale}$
 $F(\text{Anna}) = \text{Anandan}$
 $F(\text{Abigale}) = \text{Ayuta}$
 $M(\text{Abigale}) = \text{Anahita}$
 $M(\text{Ayuta}) = \text{Abeba}$
 $M(\text{Akanksha}) = \text{Abeba}$
 $M(\text{Abhay}) = \text{Akanksha}$
 $M(\text{Aeden}) = \text{Akanksha}$

$$\forall y \forall z ((x = y) \wedge (y = z)) \supset (x = z))$$

transitivity

dition, we need to assert that when the arguments of a function are equal then the functions map to the same value. Similarly, when the arguments of a predicate are equal then the two predicates are logically equivalent. We can express these for a subset that we may require³⁶.

$$\forall x \forall y ((x = y) \supset (M(x) = M(y)))$$

substitution for functions

$$\forall x_1 \forall y_1 \forall x_2 \forall y_2 ((x_1 = x_2) \wedge (y_1 = y_2) \supset \{P(x_1, y_1) \equiv P(x_2, y_2)\})$$

substitution for predicates

$$\forall x_1 \forall y_1 \forall x_2 \forall y_2 ((x_1 = x_2) \wedge (y_1 = y_2) \supset (GP(x_1, y_1) \equiv GP(x_2, y_2)))$$

substitution for predicates

Without any loss of generality we can substitute the equivalence relation in the last two with the implication relation. This is not a loss of generality as we can always substitute the implication relation for the equivalence relation if we need.

$$\forall x_1 \forall y_1 \forall x_2 \forall y_2 ((x_1 = x_2) \wedge (y_1 = y_2) \supset (P(x_1, y_1) \supset P(x_2, y_2)))$$

substitution for predicates

$$\forall x_1 \forall y_1 \forall x_2 \forall y_2 ((x_1 = x_2) \wedge (y_1 = y_2) \supset (GP(x_1, y_1) \supset GP(x_2, y_2)))$$

substitution for predicates

We can say that the question we are asking is “Does Arushi’s mother have a cousin?” which is equivalent to asking if the following is true,

$$\exists x C(M(\text{Arushi}), x)$$

Figure 12.15 gives us a graphical view of the relationships in the database.

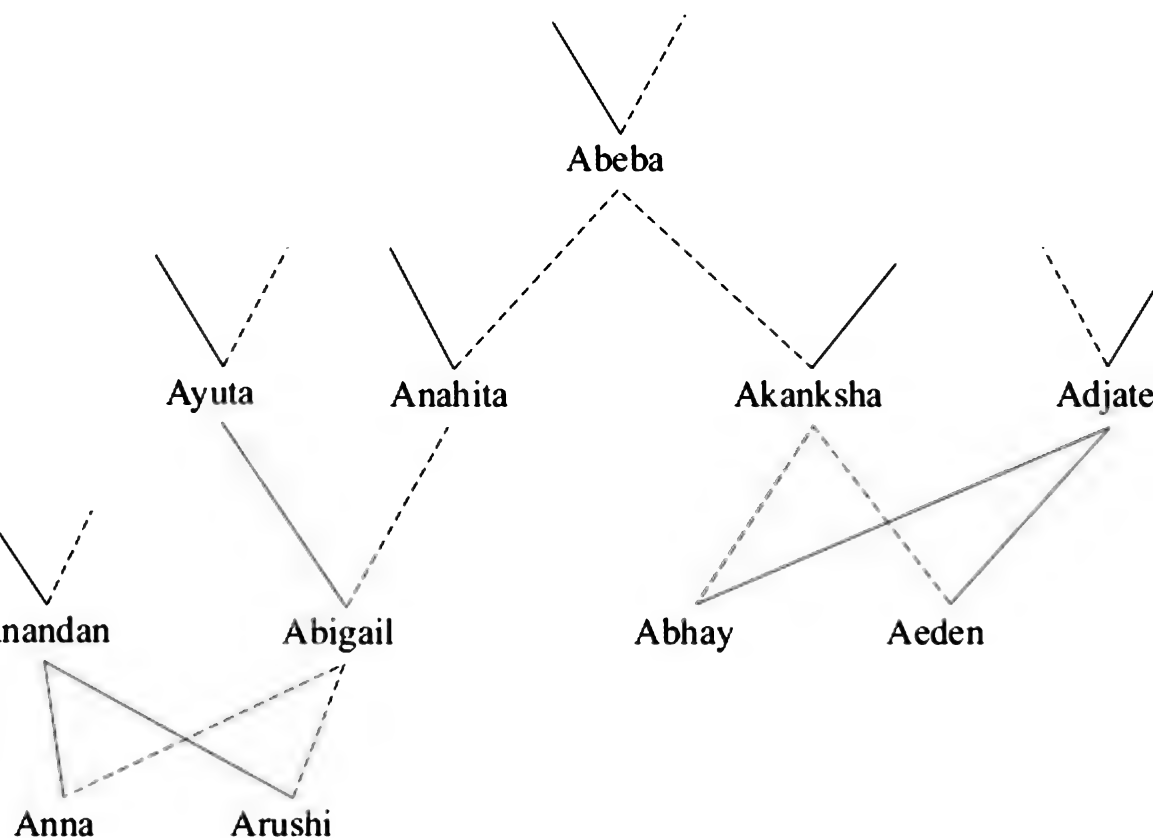


Figure 12.15 A small family data base. Dashed lines represent the mother child relation while solid lines represent the father child relation.

We can see from the figure Arushi’s mother Abigail has two cousins, Abhay and Aeden, from the data available. We can express the data base and the negated goal (along with the answer predicate) as a set of clauses in first order logic. We discard the outermost brackets when there is no ambiguity.

$$M(\text{Arushi}) = \text{Abigail}$$

$$F(\text{Arushi}) = \text{Anandan}$$

$$M(\text{Anna}) = \text{Abigail}$$

$$F(\text{Anna}) = \text{Anandan}$$

arity related axioms,

$$\begin{aligned} & \neg (x_8 = ?x_8) \\ & \neg (?x_9 = ?y_9) \vee (?y_9 = ?x_9) \\ & \neg (?x_{10} = ?y_{10}) \vee (\neg (?y_{10} = ?z_{10})) \vee (?x_{10} = ?z_{10}) \end{aligned}$$

stitution related axioms for the function M , and predicates P and GP .

$$\begin{aligned} & \neg (?x_{11} = ?y_{11}) \vee ((M ?x_{11}) = (M ?y_{11})) \\ & \neg (?x_{12} = ?x_{22}) \vee (\neg (?y_{12} = ?y_{22})) \vee (\neg (P ?x_{12} ?y_{12})) \vee (P ?x_{22} ?y_{22}) \\ & \neg (?x_{13} = ?x_{23}) \vee (\neg (?y_{13} = ?y_{23})) \vee (\neg (GP ?x_{13} ?y_{13})) \vee (GP ?x_{23} ?y_{23}) \end{aligned}$$

stitution goal along with the answer predicate is added to the set of clauses.

$$\neg (C (M \text{ Arushi}) ?x) \vee (\text{Answer } ?x)$$

is given below. The underlined predicates are the ones being resolved out in a later, usual resolution.

$$\begin{aligned} & \neg (\underline{(GP ?x_5 (M \text{ Arushi}))}) \vee (\neg (GP ?x_5 ?x)) \vee (\text{Answer } ?x) \text{ 26, 17, } ?y_5 = (M \text{ Arushi}), ?z_5 = ?x \\ & \neg (?x_{13} = ?x_5) \vee \neg (\underline{(?y_{13} = (M \text{ Arushi}))}) \vee (\neg (GP x_{13} y_{13})) \vee (\neg (GP ?x_5 ?x)) \vee (\text{Answer } ?x) \text{ 27, 2} \\ & \text{5, } ?y_{23} = (M \text{ Arushi}) \\ & \text{Abigail} = (M \text{ Arushi}) \text{ 1, 21, } ?x_9 = (M \text{ Arushi}), ?y_9 = \text{Abigail} \\ & \neg (?x_{13} = ?x_5) \vee \neg (\underline{(GP x_{13} \text{ Abigail})}) \vee (\neg (GP ?x_5 ?x)) \vee (\text{Answer } ?x) \text{ 26, 29, } ?y_{13} = \text{Abigail} \\ & \neg (?x_{13} = ?x_5) \vee (\neg (P ?x_{13} ?y_4)) \vee \neg (\underline{(P ?y_4 \text{ Abigail})}) \vee (\neg (GP ?x_5 ?x)) \vee (\text{Answer } ?x) \text{ 30, 16. } ?x_4 = \\ & \text{Abigail} \\ & \neg (?x_{13} = ?x_5) \vee (\neg (P ?x_{13} ?y_4)) \vee \neg (\underline{(MAbigale) = ?y_4}) \vee (\neg (GP ?x_5 ?x)) \vee (\text{Answer } ?x) \text{ 31, 13, } ? \\ & \text{y}_4 = \text{Abigale} \\ & \neg (?x_{13} = ?x_5) \vee (\neg (P ?x_{13} \text{ Anahita})) \vee (\neg (GP ?x_5 ?x)) \vee (\text{Answer } ?x) \text{ 32, 6, } ?y_4 = \text{Anahita} \\ & \neg (?x_{13} = ?x_5) \vee \neg (\underline{(M \text{ Anahita}) = ?x_{13}}) \vee (\neg (GP ?x_5 ?x)) \vee (\text{Answer } ?x) \text{ 33, 13, } ?x_1 = ?x_{13}, ?y_1 = \text{An} \\ & \text{Abeba} = ?x_5) \vee (\neg (GP ?x_5 ?x)) \vee (\text{Answer } ?x) \text{ 34, 7, } ?x_{13} = \text{Abeba} \\ & \text{Abeba} = (M \text{ Akanksha}) \text{ 8, 21, } ?x_9 = (M \text{ Akanksha}), ?y_9 = \text{Abeba} \\ & \neg (\underline{(GP (MAkanksha) ?x)}) \vee (\text{Answer } ?x) \text{ 35, 36, } ?x_5 = (M \text{ Akanksha}) \\ & \neg (\underline{(P (M \text{ Akanksha}) ?y_4)}) \vee (\neg (P ?y_4 ?x)) \vee (\text{Answer } ?x) \text{ 37, 16, } ?x_4 = (M \text{ Akanksha}), ?z_4 = x \\ & \neg (\underline{(M ?y_4) = (MAkanksha)}) \vee (\neg (P ?y_4 ?x)) \vee (\text{Answer } ?x) \text{ 38, 13, } ?x_1 = (MAkanksha), ?y_1 = ?y_4 \\ & \neg (\underline{(?y_4 = Akanksha)}) \vee (\neg (P ?y_4 ?x)) \vee (\text{Answer } ?x) \text{ 39, 23, } ?x_{11} = ?y_4, ?y_{11} = \text{Akanksha} \\ & \neg (\underline{(P (M \text{ Abhay}) ?x)}) \vee (\text{Answer } ?x) \text{ 40, 9, } ?y_4 = (M \text{ Abhay}) \\ & \neg (\underline{(M ?x) = (M \text{ Abhay})}) \vee (\text{Answer } ?x) \text{ 41, 13, } ?x_1 = (M \text{ Abhay}), ?y_1 = ?x \\ & \neg (\underline{((M ?x) = ?y_{10})}) \vee (\neg (?y_{10} = (M \text{ Abhay}))) \vee (\text{Answer } ?x) \text{ 42, 22, } ?x_{10} = (M ?x), ?z_{10} = (M \text{ Abhay}) \\ & \text{Akanksha} = (M \text{ Abhay}) \text{ 9, 21, } ?x_9 = (M \text{ Abhay}), ?y_9 = \text{Akanksha} \\ & \neg (\underline{(M ?x) = Akanksha}) \vee (\text{Answer } ?x) \text{ 43, 44, } ?y_{10} = \text{Akanksha} \\ & \text{Answer Abhay} \text{ 45, 9, } ?x = \text{Abhay} \end{aligned}$$

that only one of the two answers is extracted.

use of equality allows us more expressive power, but it does increase the number of inference steps substantially. One popular method is to introduce another rule to the resolution method called *Paramodulation* (Robinson and Wos, 1969). Paramodulation allows one to substitute values directly during resolution. The rule follows,

$$\begin{aligned} \text{m:} & \alpha \vee (t = s) \\ \text{d:} & \beta \vee \rho(t') \\ \text{r:} & (\alpha \vee \beta \vee \rho(s))\theta \end{aligned}$$

is the most general unifier for t and t' .

above proof for example applying the paramodulation rule to statements 27 and 1 we directly get,

$$\neg (\underline{(GP ?x_5 \text{ Abigale})}) \vee (\neg (GP ?x_5 ?x)) \vee (\text{Answer } ?x)$$

is a simpler form of statement 30!

ability however is another matter. Also, the resolution method, and any other method operating upon a matter, is not decidable. That means that the algorithm may not always terminate. When the goal is false, the proof results say that a proof can be found. However, when the goal is not true, or equivalently the negated goal is not unsatisfiable, which means they are satisfiable then the procedure may go into an infinite loop.

If the domain is finite, then the entire set of *FOL* statements can be expanded into propositional logic. This can be done by exploiting the fact that the universal quantifier is essentially a conjunction over the domain and the existential quantifier is essentially a disjunction. For example in a universe of discourse containing the three numbers {1, 2, 3}, the following statements can be expressed in propositional logic.

$\forall x \text{ Even}(x)$ can be replaced by $(\text{Even}(1) \wedge \text{Even}(2) \wedge \text{Even}(3))$

$\exists x \text{ Even}(x)$ can be replaced by $(\text{Even}(1) \vee \text{Even}(2) \vee \text{Even}(3))$

$\forall x \exists z (\text{LessThan}(x, z))$ can be replaced by³⁷,

$(\text{LessThan}(1, 1) \vee \text{LessThan}(1, 2) \vee \text{LessThan}(1, 3))$
 $\wedge (\text{LessThan}(2, 1) \vee \text{LessThan}(2, 2) \vee \text{LessThan}(2, 3))$
 $\wedge (\text{LessThan}(3, 1) \vee \text{LessThan}(3, 2) \vee \text{LessThan}(3, 3))$

that that the constituent formulas are not necessarily true.

For example like $\forall x \forall y [\text{LessThan}(\text{successor}(x), y) \supset \text{LessThan}(x, y)]$ will get replaced by

$(\text{LessThan}(\text{successor}(1), 1) \supset \text{LessThan}(1, 1))$
 $\wedge (\text{LessThan}(\text{successor}(1), 2) \supset \text{LessThan}(1, 2))$
 $\wedge (\text{LessThan}(\text{successor}(1), 3) \supset \text{LessThan}(1, 3))$
 $\wedge (\text{LessThan}(\text{successor}(2), 1) \supset \text{LessThan}(2, 1))$
 $\wedge (\text{LessThan}(\text{successor}(2), 2) \supset \text{LessThan}(2, 2))$
 $\wedge (\text{LessThan}(\text{successor}(2), 3) \supset \text{LessThan}(2, 3))$
 $\wedge (\text{LessThan}(\text{successor}(3), 1) \supset \text{LessThan}(3, 1))$
 $\wedge (\text{LessThan}(\text{successor}(3), 2) \supset \text{LessThan}(3, 2))$
 $\wedge (\text{LessThan}(\text{successor}(3), 3) \supset \text{LessThan}(3, 3))$

As we can see this can become cumbersome, and one has also to worry about definitions of functions like $\text{successor}(x)$, which is not defined for the number 3. However whenever one can convert a set of formulas into propositional logic one can rely on the fact that the proof procedures are decidable. They will terminate one way or the other. On the other hand *FOL* allows us to talk of infinite sets, and it is in reasoning with infinite sets that the undecidability lurks. The procedure may keep trying out a never ending sequence of bindings.

Let us look at an example to illustrate the fact. Given the set of (true) formulas, over the domain of natural numbers:

$\forall x \forall y [\text{LessThan}(\text{successor}(x), y) \supset \text{LessThan}(x, y)]$
 $\forall x \forall y [\text{LessThan}(x, y) \supset \text{LessThan}(\text{successor}(x), \text{successor}(y))]$
 $\text{LessThan}(0, \text{successor}(0))$

As a goal $\text{LessThan}(\text{successor}(\text{successor}(0)), \text{successor}(\text{successor}(\text{successor}(0))))$ one can indeed find a proof of the null clause, because the formula is true. This is left as an exercise. However given a goal that is false, for example “ $\text{LessThan}(\text{successor}(0), \text{successor}(0))$ ” the procedure may never end, generating a sequence of bindings as shown below,

$\neg (\text{LessThan}(\text{successor } ?x_1 \text{ } ?y_1) \vee (\text{LessThan } ?x_1 \text{ } ?y_1))$
 $\neg (\text{LessThan } ?x_2 \text{ } ?y_2) \vee (\text{LessThan}(\text{successor } ?x_2)(\text{successor } ?y_2))$
 $\text{LessThan } 0 (\text{successor } ?x_3))$
 $\neg (\text{LessThan}(\text{successor } 0)(\text{successor } 0))$
 $\neg (\text{LessThan } 0 \ 0))$
 $\neg (\text{LessThan}(\text{successor } 0) \ 0))$
 $\neg (\text{LessThan}(\text{successor}(\text{successor } 0)) \ 0))$
 $\neg (\text{LessThan}(\text{successor}(\text{successor}(\text{successor } 0))) \ 0))$

negated goal

2, 4, $?x_2 = 0$, $?y_2 = 0$

1, 5, $?x_1 = 0$, $?y_1 = 0$

1, 6, $?x_1 = (\text{successor } 0)$, $?y_1 = 0$

1, 7, $?x_1 = (\text{successor}(\text{successor } 0))$, $?y_1 = 0$

The programmer has to specify the logical relations between the input and output, and leave the task of finding the sequence of commands to the language interpreter (Kowalski, 1979a). However, there is no free lunch. We shall look at briefly, the idea of "programming" is still necessary if one is to write efficient code. The book on Prolog has been (Clocksin and Mellish, 2003). A good introduction to logic programming can be found in (Clocksin and Shapiro, 1994), an account of its theoretical foundations in (Lloyd, 1984), and a recent introduction in (Blackburn et al, 2006).

A *Horn clause* is a clause with at most one positive literal. Thus, it is of the form,

$$\neg D_1 \vee \neg D_2 \vee \dots \vee \neg D_k \vee D_{k+1}$$

It is also convenient to think of a Horn clause like the one above as a *rule*,

$$(D_1 \wedge D_2 \wedge \dots \wedge D_k) \supset D_{k+1}$$

These two forms are logically equivalent. The key feature is that a Horn clause does not allow a disjunctive head. This also means that disjunctive uncertainty cannot be expressed in Horn clause logic. One cannot express a statement like $(\text{Green}(A) \vee \text{Green}(B))$ because that is not a Horn clause. A conjunction can be incorporated into the body of the clause, and one rule to infer the individual elements, and another rule to infer the consequent. For example, if one has a rule $(A \wedge B) \supset (C \wedge D)$, one could write the rules,

$$\begin{aligned} (A \wedge B) &\supset C \\ (A \wedge B) &\supset D \\ (C \wedge D) &\supset F \end{aligned}$$

where F stands for the conjunct, or $F \equiv (C \wedge D)$.

A *fact* is a Horn clause made up of only a positive literal, like "A", and such a literal is also known as a *fact*. A clause with no antecedents is a *goal*.

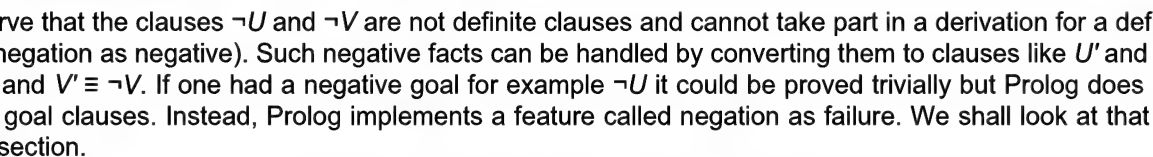
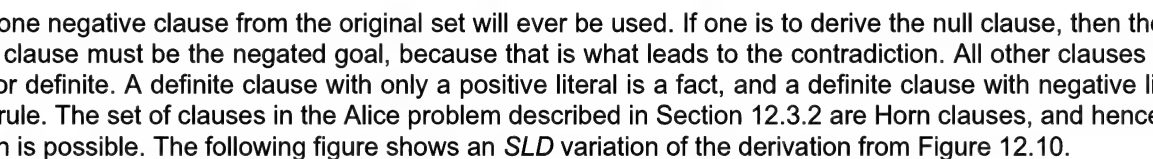
A Horn clause which has a positive literal is known as a *definite* or *positive* clause; else it is called a *negative* clause. The empty clause is a negative Horn clause. In any resolution step at least one of the parent clauses must be a negative clause.

We are interested in deriving a clause G from a set of Horn clauses S . In the resolution method we add $\neg G$ to S . If a clause C is derivable from a set of Horn clauses S , we say that $S \vdash C$.

We are given a set of Horn clauses and a negative clause C that can be derived by applying resolution steps, and we want to construct a derivation in which every resolvent is a negative clause (Brachman and Levesque, 2004). This is the goal of the derivation of the null clause as well, because it too is a negative clause. We illustrate how this can be done by taking a small example. In Figure 12.16, the derivation on the left has a positive resolvent. The derivation on the right derives the same final clause, but does not contain a positive *resolvent*. Such a derivation uses only negative clauses and one positive clause to resolve at each step. Given a derivation of a negative clause with positive resolvents, one can always transform it by eliminating the lowest positive resolvent as shown in the figure. We repeat the process till all resolvents are negative.

The figure shows two resolution trees. The left tree starts with three parent clauses: $\neg E \vee C$, $\neg F \vee \neg V \vee D$, and $\neg D \vee \neg A$. The first two are resolved to form a shaded box containing $\neg E \vee \neg F \vee D$. This shaded box and the third parent clause are then resolved to form the final clause $\neg E \vee \neg F \vee \neg A$. The right tree starts with the same three parent clauses. The last two are resolved to form a box containing $\neg F \vee \neg C \vee D$. This box and the first parent clause are then resolved to form the final clause $\neg E \vee \neg F \vee \neg A$.

12.16 The figure on the left has a positive *resolvent*, shown in the shaded box. The equivalent derivation on the right has only negative resolvents. One can transform a derivation with positive resolvents to one with only negative resolvents by picking the lowest positive resolvent and transforming it like the one above.



has been shown to be complete for Horn clauses, and hence for whatever knowledge bases one can use from there exists the possibility of constraining search to find proofs faster.

Each program is a set of definite clauses. Each negative clause represents a possible goal, shown as figure 12.17. Observe that a disjunctive goal would result in a set of negative clauses. Each call to the goal invokes a custom made sequence of inferences. In that sense a logic programming language is a goal language. The programmer does not have to specify what to do next. She has to only specify the different statements in terms of rules and facts. Combined with the fact that one can extract the

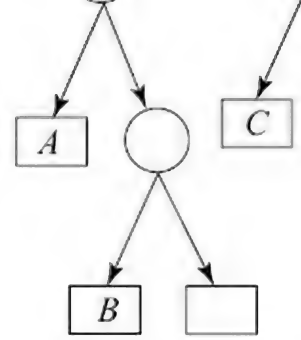
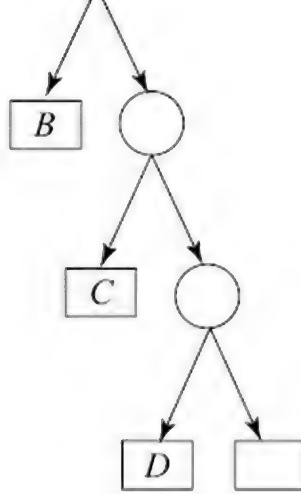
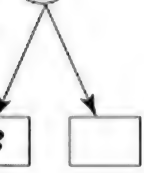


Figure 12.19 The tree structures for the lists [A B], [A B C D], and [[A B] C] respectively. The circular node is also known as the dotted pair. The left child points to the head of the list and the right child to a list that is the tail of the given list.

Internal nodes are also called *dotted pairs* (Left.Right) or *cons pairs* (cons Left Right). The left element of the dotted pair is the head of the list and the right element is a list that is the tail. The right element may be empty. We will use the notation in which “nil” is another way of representing an empty list ().

$[A\ B] = (\text{cons } A\ (\text{cons } B\ \text{nil}))$
 $[A\ B\ C\ D] = (\text{cons } A\ (\text{cons } B\ (\text{cons } C\ (\text{cons } D\ \text{nil}))))$
 $[[A\ B]\ C] = (\text{cons } (\text{cons } A\ (\text{cons } B\ \text{nil}))\ (\text{cons } C\ \text{nil}))$

Implementing a program to append two lists is basically a definition of what it means to append. The predicate $\text{append}(X, Y, Z)$ is true when Z is the result of appending lists X and Y . The following axioms capture the append relation.

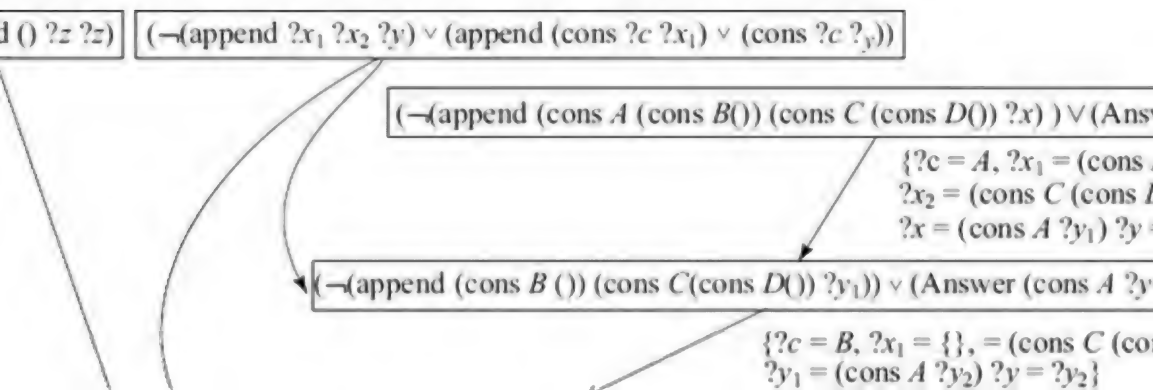
$\forall x\ \text{append}(\text{nil}, x, x)$
 $\forall x\forall y\forall z\forall c\ (\text{append}(x, y, z) \supset \text{append}(\text{cons}(c, x), y, \text{cons}(c, z)))$

The first clause in clause form with variables standardized apart may be written as,

$(\text{append } ()\ ?z\ ?z)$
 $(\neg(\text{append } ?x_1\ ?x_2\ ?y)) \vee (\text{append } (\text{cons } ?c\ ?x_1)\ ?x_2\ (\text{cons } ?c\ ?y))$

The two clauses represent a program to append two lists!

A query to this program may be made by specifying a goal. For example, we could ask whether $\text{append}([A\ B], [C\ D], X)$ is true or not³⁹. We could also plug in variables to ask queries like $\exists x\ \text{append}([A\ B], [C\ D], x)$. The last query corresponds to the standard use of an append program, but it allows us more flexibility. A derivation for the query $\exists x\ \text{append}([A\ B], [C\ D], x)$ accompanied with an answer is shown in Figure 12.20.



derivation on Horn clauses can be seen as a backward chaining process on the set of rules. A goal G is a sentence that we want to show to be true. If the goal exists as a fact then it is true by definition. Otherwise if $G \supset Q$ whose consequent matches the goal then we regress over the rule to produce a sub-goal Q to be proved. If the antecedent has more than one sentence then we add all those to the set of goals. If the antecedent is a fact then it is removed from the set. The procedure terminates when the goal set is empty. The procedure is an example of the descent of backward state space planning of Chapter 7.

Illustrate the process first with the propositional version of the Alice problem and then describe the process in general.

Given data is repeated here,

- | | |
|-------------------------------|---------------------|
| 1. $(P \wedge Q)$ | not Horn clause |
| 2. $(P \supset R)$ | |
| 3. $((R \wedge S) \supset T)$ | |
| 4. $(\neg Q \vee S)$ | |
| 5. $(\neg U \wedge \neg V)$ | not definite clause |

From these we select the ones corresponding to the definite clauses and express them as facts and rules of Horn clauses.

- $(P \supset R)$
 $((R \wedge S) \supset T)$
 $(Q \supset S)$

The required goal is T . We initialize the goal set to T , and use $\{ \}$ notation to distinguish goals from facts. The derivation mirrors the SLD resolution of Figure 12.18.

- | | |
|---------------|-----------------------|
| 1. $\{T\}$ | given goal |
| 2. $\{R, S\}$ | chaining T with 4 |
| 3. $\{P, S\}$ | chaining R with 3 |
| 4. $\{S\}$ | removing P due to 1 |
| 5. $\{Q\}$ | chaining S with 5 |
| 6. $\{ \}$ | removing Q due to 2 |

The procedure adopted is to start with the given goal (set) and reduce it to empty goal set. The goal set is reduced by adding a clause, resulting in a contradiction. Instead a separate notation for goals was introduced. The procedure continues till the goal set becomes empty.

The backward chaining step using modus ponens for FOL may be formulated as follows.

Given a goal set $\{ \dots \phi \dots \}$ containing ϕ and a rule $\beta \supset \delta$ we can define a backward chaining step as follows. If β and δ have a unifier θ then replace the sub-goal ϕ with the sub-goal $\beta\theta$ in the goal set to get $\{ \dots \beta \dots \}\theta$. The backward chaining algorithm can then be written as in Figure 12.21.

```

backwardChaining(F : facts, R : rules, g : goal)
goalSet ← (g)
proof ← ()
theta ← ()
while goalSet ≠ ()
do subGoal ← Head(goalSet)
   goalSet ← Rest(goalSet)
   if there exists a substitution beta s.t. Apply(beta, subGoal) ∈ F
   then proof ← Cons((Apply(beta, subGoal), "Premise" ), proof)
   else CHOOSE r = ((antecedents) (consequent)) ∈ R
        s.t. alpha ← Unify(consequent, subGoal) ≠ ()
        if no such rule exists
        then return FAIL

```

order is the order in which the programmer writes them! Even then the program may have to backtrack choices.

In logic programming the user does not have to worry about the order in which statements are made. It is on the burden of finding the correct clauses to resolve to the (non-deterministic) inference engine. In some engines are not (yet) smart enough to make the correct choices, and in languages like Prolog one has to order the statements. This is needed not only for efficiency in terms of number of logical inferences⁴⁰, but often also for termination. A good programmer writes programs that do not get into infinite loops. A good logic programmer orders her clauses so that the inference engine does not wander down a wild goose chase (infinite loop).

Prolog inverts the notation of writing a rule, with the consequent on the left and the antecedents on the right. The symbol “:-” to separate the consequent from the antecedents, and a comma to separate the antecedents. A problem in *FOL* written in Prolog like rules would be as follows,

```
likes(alice, math).
likes(alice, stories).
likes(X, algebra) :- likes(X, math).
goesTo(X, college) :- likes(X, algebra), likes(X, physics).
likes(alice, physics) :- likes(alice, stories).
```

Prolog uses capitalized words for variables, and words beginning with lower case letters for constants. The notation “consequent if antecedents”. For example “someone likes to go to college if she likes algebra and physics” (rule in line 4 above). The advantage of writing the consequent on the left is that given a goal the interpreter goes down the program looking at only the first predicate in each line, known as the *head* of the clause. If found either it is a fact, or a consequent of a rule. If it is a fact success has been achieved for the goal. If not, recursive calls are made with the antecedents starting from the leftmost. As shown below, given a goal the interpreter answers either with a “yes”, or variable bindings that succeed, or “no”. A query is typed in standard Prolog as follows

```
?- goesTo(alice, college).
yes

?- goesTo(X, college)
X = Alice
```

“no” means that the interpreter has been unable to prove the goal. Since backward chaining with Horn clauses is complete⁴¹, it also means the goal is not entailed by the set of definite clauses.

1 Goal Trees

The search space explored by backward chaining, and hence by *SLD* resolution with Horn clauses, is a goal tree. This is discussed in Chapter 6.

We now formulate the planning an outing problem of Chapter 6 as Prolog clauses. The task or goal is to find a plan to go out with a friend.

```
outingPlan(X, Y, Z) :- eveningPlan(X), moviePlan(Y), dinnerPlan(Z).
eveningPlan(X) :- outing(X), likes(friend, X).
moviePlan(X) :- movie(X), likes(friend, X).
dinnerPlan(X) :- restaurant(X), likes(friend, X).
outing(mall).
outing(beach).
movie(theMatrix).
movie(artificialIntelligence).
movie(bhuvanShome).
movie(sevenSamurai).
restaurant(pizzaHut).
restaurant(saravanaBhavan).
likes(friend, beach).
likes(friend, theMatrix).
```

```

likes(friend, mall), moviePlan(Y), dinnerPlan(Z)} theta = {X=mall}
"fail", moviePlan(Y), dinnerPlan(Z)} theta = {X=mall}
outing(X), likes(friend, X), moviePlan(Y), dinnerPlan(Z)} theta = { } backtrack
likes(friend, beach), moviePlan(Y), dinnerPlan(Z)} theta = {X=beach}
moviePlan(Y), dinnerPlan(Z)} theta = {X=beach}
movie(Y), likes(friend, Y), dinnerPlan(Z)} theta = {X=beach}
likes(friend, theMatrix), dinnerPlan(Z)} theta = {X=beach, Y=theMatrix}
dinnerPlan(Z)} theta = {X=beach, Y=theMatrix}
restaurant(Z), likes(friend, Z)} theta = {X=beach, Y=theMatrix}
likes(friend, pizzaHut)} theta = {X=beach, Y=theMatrix, Z=pizzaHut}
"fail"} theta = {X=beach, Y=theMatrix, Z=pizzaHut}
restaurant(Z), likes(friend, Z)} theta = {X=beach, Y=theMatrix} backtrack
likes(friend, saravanaBhavan)} theta = {X=beach, Y=theMatrix, Z= saravanaBhavan }
} theta = {X=beach, Y=theMatrix, Z= saravanaBhavan }

```

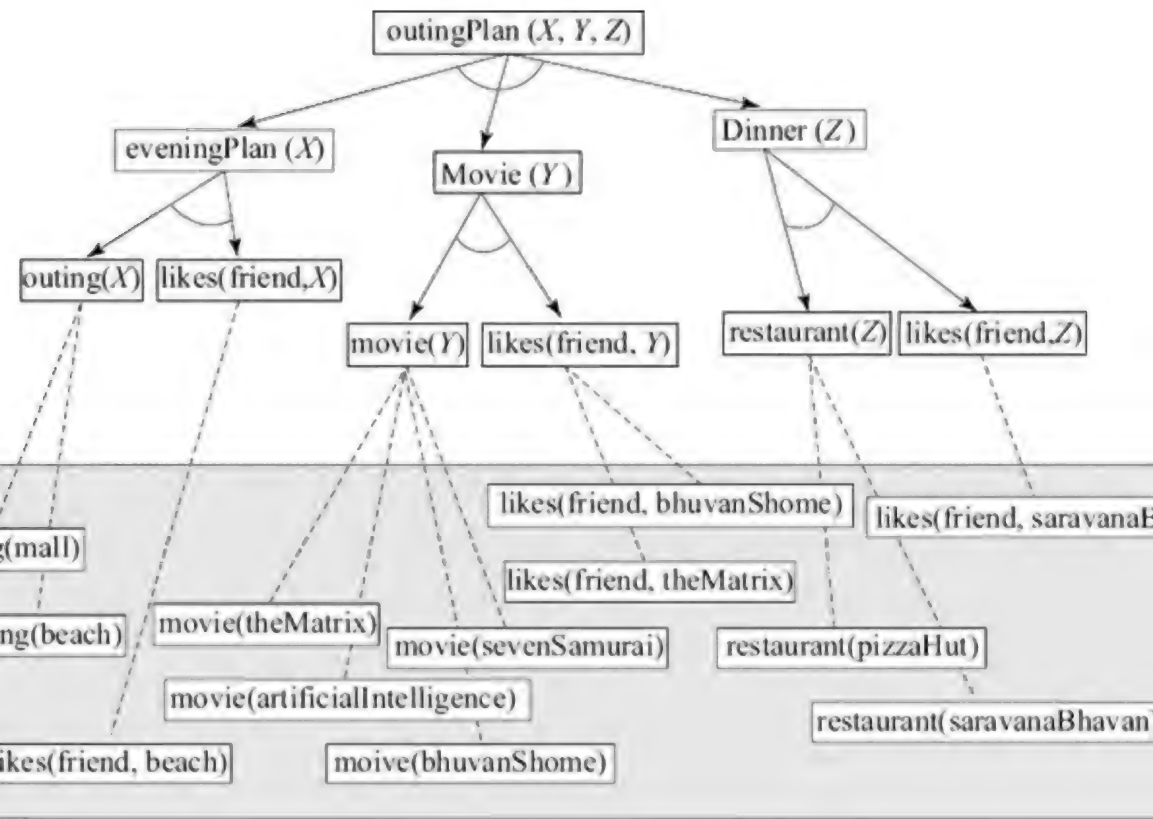


Figure 22 The goal tree explored by backward chaining is shown in solid arrows. The matching facts are shown in the boxes in the shaded area. The algorithm explores the tree in a depth first manner left to right.

What would have happened if your friend liked neither the Pizza Hut nor Saravana Bhavan? The algorithm would have backtracked and attempted the next choice of a movie. That sub-goal would eventually have succeeded with the movie *Bhuvan Shome*. Now it would go over the choices of the restaurants all over again even though we know the choices are bound to fail.

To prevent such fruitless backtracking Prolog allows the programmer to mark certain sub-goals with the `cut` operator. Using the `cut` operator, for which Prolog uses the symbol `!`, we can rewrite the rule as follows.

```
outingPlan(X, Y, Z) :- eveningPlan(X), !, moviePlan(Y), !, dinnerPlan(Z).
```

The second cut in the above rule essentially says the following: If the *eveningPlan(X)* goal and the *moviePlan(Y)* goal succeeded with some bindings of the variables *X* and *Y*, and if the *dinnerPlan(Z)* goal fails for all bindings of *Z*, there is no need to backtrack to the *eveningPlan(X)* and *moviePlan(Y)* goals to try different bindings of *X* and *Y*. The goal *outingPlan(X, Y, Z)* should report "fail". Even the next clause if there is one for the parent goal.

```

scienceFictionFilm(theMatrix).
actionFilm(whereEaglesDare).
emotionalDrama(scenesFromAMarriage).

```

3–6 define different genres of films. Lines 7–12 give some examples of the different genres. Lines 13–14 show that Balaji likes all movies except emotional dramas. The way that works is as follows. If we make the query

```

?- likesMovie(balaji, bladeRunner).
yes

```

the answer will be yes. This is because Prolog first tries rule 1, in which the subgoal *emotionalDrama(bladeRunner)* fails. Prolog backtracks and tries rule 2, which eventually triggers the rule in line 5 that succeeds. If we ask,

```

?- likesMovie(balaji, scenesFromAMarriage).
no

```

Prolog will say no. This happens because the *emotionalDrama* sub-goal in rule 1 succeeds. The inference then proceeds to the next predicate cut, which succeeds by definition, and finally to the last one that returns “fail”. At the end of the cut operator the inference abandons its search and reports “fail” which Prolog interprets as “no”. This example shows that the ordering of the clauses is critical. If the order of clauses 1 and 2 were to be changed, the result could have been different. Also note that an existential query “Does there exist a movie that Balaji likes?” is answered correctly as an exercise the reader is encouraged to modify the clauses such that this query too is answered correctly. The use of *cut* in the movie example changed the meaning of the clauses. The clauses now entail that Balaji likes all movies that are emotional dramas. Such cuts are known as *red cuts*, as opposed to *green cuts* like the ones in the sorting plan example that are only there for efficiency.

We can use the *cut-fail* combination to define the negation of a predicate as failure to prove it. If the goal fails, the negation of the goal is true. The following set of statements defines negation as failure.

```

neg(G) :- G, !, fail.
neg(G).

```

The first clause is true then due to the first line fail (no) is returned, else true (yes) is returned. We can in fact write out the second clause in one line as follows,

```

likesMovie(balaji, X) :- movie(X), neg(emotionalDrama(X)).

```

This says that Balaji likes all movies as long as they are not emotional dramas.

The use of negation as failure provides us a useful tool for defining compliments of sets. For example we can define the set of composite numbers as follows,

```

compositeNumber(N) :- neg(primeNumber(N)).

```

```

weakStudent(S) :- neg(brightStudent(S), neg(averageStudent(S)).

```

We will not get into the intricacies of Prolog, but only make the observation that while it may be sufficient for many purposes, the relations in a logic programming language, it becomes necessary to do so with extreme care if we want to build efficient systems. So much so that writing the rules in a language like Prolog involves as much care as writing programs in other languages. We illustrate this point in closing with presumably the most common non-trivial operation, that of sorting a list.

The following program describes perfectly well what it means to sort a list (see also (Lloyd, 1984)).

```

sort(X, Y) :- permutation(X, Y), sorted(Y).
permutation([], []).
permutation(cons(X, Y), cons(U, V)) :- remove(X, cons(U, V), Z), permutation(Y, Z).
remove(X, cons(X, Y), Y).
remove(X, cons(Z, Y), cons(Z, U)) :- remove(X, Y, U).
sorted([]).
sorted(cons(X, _)).

```

append(SortedSOE, cons(Xhead, SortedB)).

quicksort((),()).

partition(Pivot, cons(HeadList, TailList), cons(HeadList, Sm), Bg) :-

SmallerOrEqualThan(HeadList, Pivot), partition(Pivot, TailList, Sm, Bg)

partition(Pivot, cons(HeadList, TailList), Sm, cons(HeadList, Bg) :-

LessThan(Pivot, HeadList), partition((Pivot, TailList, Sm, Bg).

partition(Pivot, (),(),()).

Line 1 says that partition the tail of the given list into smaller and bigger elements by choosing the head as pivot, recursively (quicksort) the two lists, and append the results, with the pivot inserted between the two. Line 2 is the base case to end the recursive calls. Line 3 picks the first element if it smaller or equal, and line 4 makes a recursive call to partition. Line 5 ends the recursive calls to partitioning.

Generally, it is a good practice to write the base clauses in recursion first. Sometimes it is necessary for correctness. For example the following definition of natural numbers works only when the clauses are written in this order. Assuming that the successor function is defined over the entire set of positive and negative numbers.

naturalNumber(0).

naturalNumber(successor(N)) :- naturalNumber(N).

In the quicksort example, though we have written the base clauses after the rules. This has been done because at the time the goal will not have empty lists as arguments, and the chosen order saves on calls to unify. It is because when the base clause does occur, the goal will not match the consequent part of the rule and the next clause will be tried next.

Second Order Logic

The move from propositional logic to first order logic involved the introduction of variables and quantifiers over individuals. In propositional logic, the smallest unit was the proposition. But with the hindsight of *FOL* one could view first order logic as employing predicates with constants as arguments. For example the statement “Socrates is mortal” can be thought of as “mortal(Socrates)”. The use of quantifiers in *FOL* enabled one to talk of predicates over individuals, some assignments to variables or over all assignments.

In *2nd Order Logic* a predicate symbol stood for a specific relation on the domain. One could think of a second order logic as allowing variable predicates. Given that a predicate of arity N is a subset of the D^N , where D is the domain, the set of such subsets is the power set of the cross product. This gives us a glimpse of the number of possible predicates a variable can take.

Gödel showed in his Incompleteness Theorem that a second order or more powerful logic cannot be both consistent and complete at the same time. The proof of the theorem revolves around the construction of self-referential sentences that can neither be true nor false. While his proof is quite complex we can get an insight into the Incompleteness Theorem by trying to construct a machine that can talk about itself as shown by the logician Putnam (1992). The machine operates on a vocabulary $\{\neg, P, N, (,)\}$ and operates as follows.

An expression X is a non-empty string on the alphabet. We say that the expression is printable (= provable) if the machine can print it. We assume that the machine is complete in the sense that it will eventually print any printable expression.

The norm of an expression X is defined as $X(X)$. Let us interpret P as “printable”, and N as “the norm of X is not printable”. The norm of the language (=logic) is an expression of one of the following four forms. We also describe along with the sentences.

$P()$: X is printable.

$P(X)$: The norm of X is printable, or $P(X(X))$.

$\neg P(X)$: X is not printable.

$\neg P(X(X))$: The norm of X is not printable, or $\neg P(X(X))$.

If X is an expression. A sentence is true when what it asserts (about its domain the machine) is true. $\neg P(X)$ is true if and only if X is not printable. The statements in the (logic) machine are talking about itself, as to what the machine can print and what it cannot print.

If the machine is sound and prints “ $P(X)$ ” at any stage it means that $P(X)$ is true, this in turn says that the norm of X is printable. So if the machine prints $P(X)$ it will at some time print X as well. And if it prints “ $\neg P(X)$ ” then it

Discussion

reasoning is concerned with making the implicit explicit. Given a set of statements, there are others entailed by the explicit statements. The exteriorization of the implicit is done by means of rules. The question of what rule to apply and when is critical for the effective functioning of a logic machine. It is to apply a rule of inference when one can. This is called assertion time inference or forward chaining (K and McDermott, 1985). If one applies a rule as soon as its antecedents are available, in a database, then one ends up with a larger database of statements containing all entailed statements in an explicit form. This data base would just be a matter of lookup. Of course this can only be done in domains where the set of statements is finite.

Another strategy is to apply rules in a lazy fashion, only when there is a goal that needs to be evaluated. This is called query time inference or backward chaining. Backward chaining is the more commonly used approach. It is goal directed. One makes inferences only to test the given goal statements, even though the response is often larger than in the assertion time inference for smaller databases. An advantage of query time inferences is that it makes only the inferences that are required, and that the data base does not bloat up with true but unused statements.

Both forward chaining and backward chaining are unable to find proofs where reasoning by cases is required. Gödel's resolution method is not only complete for *FOL*, it also has the added advantage that it works for all forms of inference. The resolution method was been very popular for writing automatic theorem provers, though its efficiency can be quite high. In practice one has to attenuate the complexity by carefully feeding in the rules. Once we do that, our theorem provers can treat logic as a programming language. The *SLD* resolution method turns out to be backward chaining in disguise, and is the basis of the programming language Prolog. The decidability of *FOL* is reflected in the fact that one can write programs in Prolog, and other languages that may not terminate, that never terminate.

Formal reasoning is concerned with making incontrovertible inferences. We can build sound and complete logic machines. But Gödel's incompleteness showed that we cannot build more powerful logic machines that are both sound and complete. But is that a roadblock for artificial intelligence? After all human beings are able to take higher order statements like defining the principle of mathematical induction⁴². The counter question we can ask is: can human beings be consistent? Perhaps if we can give up the requirement of consistency we might be able to build machines that can stumble upon, and serendipitously hold on to, a higher order truth.

Formal reasoning is important. There is no doubt about it. But is completeness equally important? After all, we have not been able to decide the Goldbach conjecture, written in a June 7, 1742 letter to Euler, and which states: "It seems that every number that is greater than 2 is the sum of three primes" (see for example (Guy, 2004)). It took us 357 years to find a proof for Fermat's last theorem (Wiles, 1995). And yet we are an intelligent species. Perhaps it is time of turning our focus onto our own intelligence (see (Hofstadter, 2007)) for an insightful exploration.

Artificial intelligence requires much more than logically reasoning. It requires the ability to imagine, to create, to learn, to fantasize, to plan, to perceive, make music and many other things. And it does not have to be the same for everyone.

Exercises

Assign truth values to the following statements.

- The Mount Everest is not in India.
- The tomato is a vegetable.
- The following sentence is false.
- The preceding sentence is true.
- If the Moon is made of cheese the Earth is round.
- The tomato is a fruit.
- The butterfly is an insect.

Show that all the rules in Figure 12.6 are sound.

Which of the following rule of inference a valid one?

and β

Give examples to illustrate all the valid forms of the Aristotelian syllogism.
 the following rule (from Section 12.6) a valid rule of inference?

From: β
 and δ
 and: $\frac{\alpha \vee \gamma}{(\beta \wedge \alpha) \vee (\delta \wedge \gamma)}$.
 Infer

Write a backward reasoning algorithm for propositional logic.

Show that $((R \vee Q) \wedge (P \vee \neg Q)) \equiv ((R \vee Q) \wedge (P \vee \neg Q) \wedge (R \vee P))$ is a tautology. Hint: To show this

Show that $\alpha \supset \beta$ and $\beta \supset \alpha$.

Find a resolution refutation of the problem in Figure 12.10 that uses the set of support strategy.

Give counter examples to show that the following equivalences are not tautologies

$\exists x A(x) \wedge \exists x B(x) \equiv \exists x (A(x) \wedge B(x))$

$\forall x A(x) \vee \forall x B(x) \equiv \forall x (A(x) \vee B(x))$

If we want to assert "Some men are mortal" the correct formalization is $\exists x (\text{Man}(x) \wedge \text{Mortal}(x))$, and not $\exists x (\text{Man}(x) \supset \text{Mortal}(x))$. One way to verify that the logic sentence says what is intended is to negate it and check that the negation reads correctly. Negate the two formalizations, and move the negation sign inside. How do the two negated sentences differ?

The resolution proof for the Alice problem in Figure 12.14 conforms to the unit clause strategy. But it does not conform to the set of support strategy. Produce a resolution proof for the same problem that conforms to the set of support strategy.

Use natural deduction on the detective problem of Section 12.8 to derive the formula "Culprit(Butler)". Apply resolution method to the above problem with the goal "Culprit(Butler)".

Is the following set of sentences⁴³ satisfiable? If yes, give a model for the sentences. If no, give a justification.

A father and his son were walking along the road when they met with an accident. The father died at the spot and the son was rushed to the hospital. When he was brought to the operating table the surgeon refused to operate upon him saying "I cannot operate upon this boy. He is my son".

Forward chaining and backward chaining are unable to handle rules with disjunctive consequents. What about disjunctive antecedents in rules? Can we handle rules of the kind $(\text{if } (p \vee q) \text{ then } r)$? Justify your answer.

Clarish said "The resolution method is semi-decidable. If the input formula is unsatisfiable the theorem prover will halt. If not, it could loop for ever." To which Sneha responded, "I will run two programs in parallel. To one I input the formula A, and to the other $\neg A$. One of them is bound to halt. So I will know if A is unsatisfiable or not." What do you think?

Given a set of FOL formulas in the prescribed form read the formulas from a text file and convert each into clause form. For each predicate and function add the appropriate equality axioms and convert them into clause form. Choose an appropriate naming convention for Skolem constants and functions.

Implement the Unification algorithm. For a given set of formulas in the clause form generated by assignment, implement the resolution method. Allow the user to choose between a set of strategies. Display the results in graphical form.

Backward Chaining. Implement a Prolog like backward chaining system, using a depth first strategy. Allow the user to load a program from a keyboard or file. The user should be able to save a program and read it from a file. Accept input from keyboard. Show the final proof for the goal (if possible graphically as a tree). One should be able to find more than one solution, on giving a goal. Allow the use of Cut.

Find the Most General Unifier for the following sets of formulae $\{a, b\}$, $\{c, d\}$, and $\{e, f\}$

$(\text{pays } ?X ?Y \text{ money}) \wedge (\text{gives } ?Y ?X ?\text{Object})$

$(\text{pays } (\text{brother ramesh}) (\text{uncle } ?Z) \text{ money}) \wedge (\text{gives } (\text{uncle } (\text{cousin } ?U)) (\text{brother } ?U) \text{ book})$

$p(a, x, f(g(y)))$ where a is a constant

$p(z, h(z, w), f(w))$

$p(a, x, f(g(x)))$ where a is a constant

$\text{Agent}(Z_1, \text{amar}) \vee \text{Agent}(Z_1, x_1) \vee \text{Agent}(x_1, Z_1)$

$\text{Agent}(Z_2, \text{father}(Z_2)) \vee \text{Agent}(Z_2, \text{amar})$

$\text{Agent}(\text{father}(Z_3), Z_3) \vee \text{Agent}(Z_3, \text{amar})$

Add the appropriate equality axioms and show using the Resolution method that the following set of statements is inconsistent. *MM* and *I* are constants, while *PM* and *MPP* are functions.

$MM = PM(I)$

$\forall c (MPP(c) = PM(c))$

$MM \neq MPP(I)$

Given the three statements in the preceding exercise show that the following statement follows, "All people are honest"

Reformulate the family database of Figure 12.15 without using equality. Generate a resolution proof to answer to "Does Arushi's mother have a cousin?" How do you compare the two proofs, the one with and the one without?

Add gender data to the database and define additional relationships for the above problem – Uncle, Aunty, Wife, Husband, Ancestor, Brother, Sister, Niece, Nephew, Grandson, and Grandmother.

The definitions of the family relationships in the family database allow for a person to be defined as her/himself or cousin.

$\forall x \forall y ((M(x) = M(y)) \supset S(x, y))$

$\forall x \forall y \forall z (GP(x, y) \wedge GP(x, z)) \supset C(y, z)$

Modify the above clauses so that queries like $\exists x C(\text{Abhay}, x)$, and $\exists x S(\text{Abhay}, x)$ will now answer $x = \text{Abhay}$. Apart from modifying the definitions of siblings and cousins do we need to add anything else?

Given the set of (true) formulas, over the domain of natural numbers,

$\forall x \forall y [\text{LessThan}(\text{successor}(x), y) \supset \text{LessThan}(x, y)]$

$\forall x \forall y [\text{LessThan}(x, y) \supset \text{LessThan}(\text{successor}(x), \text{successor}(y))]$

$\forall x \text{LessThan}(0, \text{successor}(x))$

Using the resolution refutation method that the goal

$\text{LessThan}(\text{successor}(\text{successor}(0)), \text{successor}(\text{successor}(\text{successor}(0))))$ is true.

Add the following set of statements to the above problem statement,

1 = successor(0)

2 = successor(1)

3 = successor(2)

4 = successor(3)

Use the paramodulation rule to derive the null clause with the goal $\text{LessThan}(2, 3)$.

Write a set of logic clauses to compute the factorial function.

Which of the following statements are true and which are false? Give a proof for the true statements and a counterexample for each of the false ones.

$(\exists x P(x) \vee \exists x Q(x)) \supset \exists x (P(x) \vee Q(x))$

$(\exists x P(x) \vee \exists x Q(x)) \supset \exists x (P(x) \vee Q(x))$

$\exists x (P(x) \vee Q(x)) \supset (\exists x P(x) \vee \exists x Q(x))$

$\forall x (P(x) \vee Q(x)) \supset (\forall x P(x) \vee \forall x Q(x))$

$(\forall x P(x) \wedge \forall x Q(x)) \supset \forall x (P(x) \wedge Q(x))$

$\forall x (P(x) \wedge Q(x)) \supset (\exists x P(x) \wedge \exists x Q(x))$

$\forall x (P(x) \vee Q(x)) \supset (\exists x P(x) \wedge \exists x Q(x))$

Consider the following facts—*Shiva, Gopal and Madhu are people. Shiva likes all kinds of food. Apples and Chicken is food. Anything anyone eats and is not killed as a result is food. If you are killed you are not a person.*

$R1: ((\text{above } ?x ?y) \wedge (\text{above } ?y ?z)) \supset (\text{above } ?x ?z?)$
 $R1': ((\text{above } ?x ?y) \wedge (\text{on } ?y ?z)) \supset (\text{above } ?x ?z?)$
 $R1'': ((\text{on } ?x ?y) \wedge (\text{above } ?y ?z)) \supset (\text{above } ?x ?z?)$
 $R2: (\text{on } ?x ?y) \supset (\text{above } ?x ?y)$

logic program to determine whether a formula of the kind (on block-A block-B) is true. Which subset of the rules given above is the most efficient ? What is the required order of rules ?

Given the database

$\{(\text{on } A B), (\text{on } B C), (\text{on } C D), (\text{on } E F), (\text{on } F G), (\text{on } G H),$
 $(\text{color } A \text{ blue}), (\text{color } B \text{ green}), (\text{color } C \text{ blue}), (\text{color } D \text{ yellow}),$
 $(\text{color } E \text{ brown}), (\text{color } F \text{ white}), (\text{color } G \text{ yellow}), (\text{color } H \text{ green})\}$

that the following sentence is true using backward chain with your choice of rules in the above problem.

$\exists x \exists y (\text{above } x y) \wedge (\text{color } x \text{ green}) \wedge (\text{color } y \text{ yellow})$

Solve the above problem using the resolution refutation method.

Given the following,

$\forall x \forall y [\text{SomeP}(\text{someF}(x), y) \supset \text{SomeP}(x, y)]$

domain containing only the constant λ and the function *someF* which of the following is a tautology? Solve using the resolution method?

$\text{SomeP}(\lambda, \lambda)$
 $\neg \text{SomeP}(\lambda, \lambda)$

four derivation steps. What does your answer imply for the soundness and completeness of the resolution method for FOL?

Given a database of Parent(parent, child) statements write an efficient Prolog like program to answer the query Ancestor(ancestor, descendant) for a country where every family has exactly one child. Assume the arguments to the query are constants from the domain. What is the worst case complexity of executing the program?

Express the following sentences in FOL.

An honest politician has given a promise he keeps the promise. If a party has given a promise, and a person is a leader of the party, then that means the person has given the promise. If a person keeps his promise, and he is the leader of a party then the party keeps the promise. If a person is the leader of a party then the person is a politician. Party 'JBCD' made a promise 'P1'. 'JBCD' did not keep the promise 'P1'. The leader of party 'JBCD' is 'KMS'. 'KMS' is a person.

Express the following predicate schema

$P(x) - x$ is a person, $\text{LoP}(x, y) - x$ is the leader of party y ,
 $H(x) - x$ is honest. $\text{Pol}(x) - x$ is a politician,
 $\text{GiveP}(x, y) - x$ gives promise y , $\text{KeepP}(x, y) - x$ keeps promise y .

show that the following set of formulas is unsatisfiable.

$\forall x (\text{Bird}(x) \supset \text{Flies}(x))$
 $\forall x (\text{Penguin}(x) \supset \text{Bird}(x))$

$\text{Bird}(\text{peppy}) \wedge \neg \text{Flies}(\text{peppy})$

Show that the following set of formulas is unsatisfiable.

$\forall x (\text{Bird}(x) \supset \text{Flies}(x))$
 $\forall x (\text{Penguin}(x) \supset \text{Bird}(x))$
 $\exists x (\text{Penguin}(x) \wedge \neg \text{Flies}(x))$

A machine is something that operates mechanically.

Ignore here rumours and other means of speculation that may be unreliable.

A *statement* or an *assertion* or a *sentence* is something that can in principle be assigned a value *true* or *false*. We will use the three terms interchangeably. For example “White wins in Chess”, or “The Earth is flat” or “The Earth is flat then the Moon is made of green cheese”. We do not know the truth value of the first statement, the second is *false* and the third *true*.

(Bogomolny, 2008) for 78 different proofs of the Pythagorean Theorem.

Brains are presumably the seat of the mind. Human beings have unusually large brains, with the brain being 1:50, as compared to other mammals 1:180; birds 1:220; reptiles 1:1500; and fish 1:5000. The brain also consumes a disproportionately large 25% of the body's energy.

We must remember that this is only a thought experiment. Searle is only imagining a situation in which a program which makes it seem that the person sitting inside can process Chinese. See (Searle, 2009) for a counter argument.

Read about Gottfried Leibniz. From Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/Gottfried_Leibniz

We will revisit this idea when we discuss Conceptual Dependency theory of Roger Schank.

De Morgan's laws: $\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$ and $\neg(P \vee Q) = (\neg P \wedge \neg Q)$.

See also http://en.wikipedia.org/wiki/Modal_logic

It is impossible for the same thing to be both affirmed and denied of the same thing at the same time and in the same way” (*Metaphysics*. IV.3, 1005b 19–20)

There are conflicting reports of Plato's year of birth.

<http://en.wikipedia.org/wiki/Orgon>

Remember the vowels that occur in each mnemonic.

http://en.wikipedia.org/wiki/Aksapada_Gautama

Different schools offer different explanations for errors in such knowledge, as when we see a snake and mistake it to be a snake.

The five elements are earth, water, light, air and ether.

Note that we have omitted some brackets and have combined multiple sentences with the *And* connective. This often does that when it does not cause any ambiguity.

Consider the case where premise is P and the conclusion $(\neg Q \vee Q)$.

Strictly speaking a substitution step $Q = \neg Q$ has to be applied before disjunctive syllogism is applicable.

It is also commonplace to refer to the set of statements as a database of facts (and rules).

We have assumed that the required instances of rules are readily available for the sake of simplicity.

A valuation can make $(S \wedge \neg S)$ true and therefore also the CNF formula containing it. Syntactically

we reduce the formula to false by first applying the substitution $(a \wedge \neg a) \equiv \text{false}$ followed by

application of the substitution rule $(a \wedge \text{false}) \equiv \text{false}$

In first order resolution the procedure may go into an infinite loop.

There could be other quantifiers as well, for example “there exists exactly one x such that...”, but more

with only these two.

constant can also be thought of as a function of arity 0, so the two are in fact similar.

When you have eliminated all which is impossible, then whatever remains, however improbable, must be the truth.

' - to quote Sherlock Holmes from The Adventure of the Blanched Soldier, by Sir Arthur Conan Doyle.

In general the functions and predicates can have an arbitrary number of arguments.

map LessThan(x, y) to $(x < y)$ in the domain.

for example <http://en.wikipedia.org/wiki/Prolog>

use the square bracket list notation in text for ease of reading, and the cons notation in the logic code.

In the early eighties the Japanese government embarked upon the Fifth Generation computing project.

At the heart of the machine was a logic programming system (Feigenbaum and McCorduck, 1983).

Performance was to be measured in LIPS (Logical Inferences Per Second).

Even with forward chaining, it is not complete for general FOL.

$((P(0) \wedge \forall i (P(i) \supset P(i + 1))) \supset \forall i (P(i)))$

is a well known puzzle. The question assumes that the surgeon is speaking the truth.

Concepts and Language

Chapter 13

The language of FOL allows us to create a representation of the elements of a domain or universe of discourse, and the relations between them. Over this domain a set of unary predicates define categories and properties, for example Man, Mortal, Student, and Bright¹. The predicates define the base *ontology* in the philosophical sense, as a *study of being* or existence. In that sense a predicate defines a *category* for some agent that wants to *reason about* the domain. We are not concerned here with the fundamental questions posed by Heidegger as to “*what is existence?*” or the “*meaning of Being*” (Heidegger, 1927), but rather what Sartre calls *consciousness* which “*is in a state of cohabitation with its material body*” (Sartre, 1943). One could take this to be a phenomenological attitude where we are concerned with *knowing about* existence as intimately tied to existence itself. Willard Van Orman Quine settled the issue as far as logic and knowledge representation is concerned when he said that “*to be is to be the value of a quantified variable*” (Quine, 1990). Then if one says for example “*for all x...*” then whatever binds to *x* as a value exists.

The notion of an apple can be defined simply as a subset of the domain that are apples, and membership to which is characterized by satisfying the predicate *Apple*. If *Apple(element291)* is *true* then *element291* is an apple. Our interest is in representation of the domain and being able to reason about the domain. If we need to *reason* about apples then we would need to associate apples with other predicates, such as apples are food, apples are (often) red, and so on.

The predicates one has in one's vocabulary are a reflection of how one *knows* the world. In computer science terminology an ontology is a formal specification of concepts and the relations between them. Tom Gruber puts it succinctly: “*An ontology is a specification of a conceptualization*” (Gruber, 1993).

Given a domain of objects, the *basic concepts* are the categories defined by unary predicates. The relations between objects are defined by the higher arity predicates, and by means of logical relationships. The first task in knowledge *representation* is to define these predicates. More complex conceptual structures can be defined in terms of simpler ones.

The many natural languages invented by human societies are ontological in the sense that they provide labels (words) for concepts that their users are dealing with. Obviously, a language will be useful (for communication) only if it is *shared*. In terms of the predicates of FOL this implies that they refer to the same subsets of the domain for different people or agents. For most of us this happens naturally since we learn a language from the people around us. A human child acquires language during its increasing interaction with the world. It is helped in this process by other more knowledgeable humans using the same language itself as a medium. Thus a child may learn to point at a four legged creature and call it a “doggy” and at a fruit and call it an “apple”. But gradually, it acquires words for “cows”, “horses”, “pears” and “guavas”, and learns to distinguish them conceptually as well as linguistically. It also learns qualifiers like “big” and “small”, and over a

period of time acquires a larger and larger vocabulary to address a larger and more refined set of concepts.

People learn and grow their ontologies (by this we mean concepts and relations between them). Starting with a few concepts they refine them more and more and the ontology begets finer and finer concepts. A child may begin by calling all four legged creatures dogs, and then gradually over a period of time refine her knowledge of the animal world. Biology students take this knowledge to a much greater level of detail, botanists have a large ontology for plants, a car mechanic knows a lot about engines and gearboxes, an architect thinks in terms of spaces and materials, a musician lives in a world of notes and melodies. The (conceptual) world of the kid living in Missouri by the Mississippi River would be entirely different from the world of a mountain boy from the Kangra valley in Himachal Pradesh, which is different from that of a Griqua child in the Kalahari desert. The Malayalis of Kerala have a multitude of names for different types of what many of us simply call a “banana”, while the Inuits of Greenland have a bewildering variety of names for different kinds of what most of us simply call “snow”². The language that a society evolves is dependent on the concepts that the society needs to articulate.

We all live in different “worlds” on this Earth. *Our world* is the mental spaces we create in our minds. What we have in our heads is knowledge about *our* world, or as some might say what we have in our heads is our world. The world is what we *imagine* it to be.

Yet each of us manages to pack in a large amount of conceptual and factual knowledge that somehow springs to the fore whenever needed.

13.1 The Conceptual Domain: The Ontological Base

The comment from Quine gives us a clue of what can exist in a domain, what can be the value of a variable. The domain in a powerful enough system must contain not only the “real” elements that one is reasoning about, but also *reified* elements that are *created* for the complex process of reasoning. We tend to think of a person as a unit that exists as a whole. But a person is made up parts. We say “my brain” or “my heart”, but the “I” that we use refers to a reified element. Our brains and hearts are in turn made up of parts, and we can do this process of deconstruction till we are talking about individual atoms. Or even subatomic particles. A person is just a collection of a vast number of atoms, estimated, for an adult, to be around 10^{27} . Surely we cannot reason with all 10^{27} of them, and we *think* in terms of “hands” and “feet” and “arms” and “legs” of a “person”.

Predicates relate to concepts, since they are an abstract description or characterization of a subset of the world. Let us assume for the sake of illustration that we *think* of the “real” domain as consisting only of humans, other life forms, and some physical objects. We look at some predicates we need to define in order to be able to reason about the tasks that an agent might want to do. Most of the predicate names that we will begin with are taken from a natural language, in our case English. This is because language has been the medium of knowledge exchange, and it is easiest for humans to deal with. But we shall also try and look at representations from a language independent perspective, independent of the grammatical rules of natural languages.

13.1.1 Categories and Relations

The simplest kinds of concepts are categories, defined by unary predicates. The

following are examples some of which we are already familiar with. We give an English language definition alongside.

Man(X)	: X is a man
Human(X)	: X is a human being
Block(X)	: X is a block
Prism(X)	: X is a prism
Fruit(X)	: X is a fruit
CitrusFruit(X)	: X is a citrus fruit

Each such predicate is interpreted in the domain as a subset of the domain, and each defines a category. A category may be related to other categories. Such relations are captured by sentences of *FOL*. For example,

$\forall x (\text{Man}(x) \supset \text{Human}(x))$: all men are human beings
$\forall x (\text{CitrusFruit}(x) \supset \text{Fruit}(x))$: all citrus fruits are fruits
$\forall x (\text{CitrusFruit}(x) \supset \neg \text{Human}(x))$: all citrus fruits are non-human
$\exists x (\text{Man}(x) \wedge \text{Bright}(x))$: some men are bright

Higher arity predicates allow us to capture relations between things. For example,

Friend(X, Y)	: X is a friend of Y
On(X, Y)	: X is on Y
Uncle(X, Y)	: X is an uncle of Y
Divides(X, Y)	: (number) X divides (number) Y

Formally a binary relation corresponds to a subset of $D \times D$, the cross product of the domain with itself. This allows us to define relations like “divides” to be applicable only to numbers by choosing the appropriate subsets in the Interpretation mapping (see Chapter 12). However, relations are rarely stored as *extensions*, and thus a more circumspect definition is usually required. One may employ a *typed* or *many sorted* logic, in which the elements belong to different *sorts* or *types*, and relations are defined over members from specific types.

Logic also provides us a mechanism for *defining* categories as *intensions* without having to explicitly mark the membership in the Interpretation mapping (Sowa, 1984). This is particularly useful when the domain size is very large, or even infinite. Thus we can define new categories from existing ones, for example,

$$\begin{aligned} \forall x (\text{Mother}(x) &= \exists y \text{Mother}(x, y)) \\ \forall x (\text{PrimeNumber}(x) &= \neg \exists y (y \neq 1 \wedge y \neq x \wedge \text{Divides}(y, x))) \end{aligned}$$

In the following chapter we shall look at a *Description Logic* that allows us to define new categories and the relations between them in a succinct manner that is also amenable to tractable reasoning. We can, and most often do, define relations as intensions as well. For example,

$$\begin{aligned} \forall x \forall y (\text{GrandParent}(x, y) &= \exists z (\text{Parent}(x, z) \wedge \text{Parent}(z, y))) \\ \forall x \forall y (\text{Mother}(x, y) &= (\text{Parent}(x, y) \wedge \text{Female}(x))) \\ \forall x \forall y (\text{Ancestor}(x, y) &= (\text{Parent}(x, y) \vee \exists z (\text{Parent}(x, z) \wedge \text{Ancestor}(z, y)))) \end{aligned}$$

The last one is a recursive definition.

13.2 Reification

So far we have only made an attempt to describe the elements of the domain and relations between them. We have not talked of *change*. We have assumed our domain to be like the domain of mathematics, where statements are always *true* or always *false*, even though we may not know about some as to what the case is. Very often in the real world one has to describe and reason about situations involving change in which agents act and events occur.

In natural languages we associate verbs with actions, and sentences with events. In FOL we can also follow an approach in which predicates stand for verbs, and thus describe actions. For example we might say,

Hit(Azizi32, Anuun12) ³	: the act of Azizi hitting Anuun
Unstack(R2D2, Block27, Block21)	: the act of R2D2 unstacking Block27 from Block21

If we look at the blocks world (see Chapter 7 on Planning) we have to worry about changing facts. For the action “*Unstack(R2D2, Block27, Block21)*” to have happened certain preconditions should have been true, for example *On(Block27, Block21)*, and certain post conditions become true after the action, for example *Clear(Block21)*. If an action actually happens we say an event has occurred. Formulas like *On(Block27, Block21)* are no longer *true* at all times but may have their truth values fluctuate between *true* and *false*. We call such formulas *asfluents*. The approach taken by the planning algorithms is to keep only the current state facts in the database, adding new ones and deleting the ones no longer true. That is sufficient for the task of planning. But if one wants to reason explicitly about change then fluents must be associated with time information. We shall look at reasoning about events a little later in the chapter.

The definition of an action predicate (or a planning operator) usually ignores the temporal aspect of the action. Natural languages, on the other hand, are finely tuned to dealing with tense and modality. If we define the predicate *Hit(X, Y)* to stand for “*X hit Y*” then what about the other tenses that language allows us, like “*X will hit Y*”, or “*X is hitting Y*”, or “*X was hitting Y*”; or “*X wants to hit Y*”, or “*X had planned to (but did not) hit Y*”. As we will see these and other issues can be tackled if we admit into our domain the abstract element standing for an action or an event. This is known as *reification*. If we say that action “*a = Hit(Azizi32, Anuun12)*” and add it to the domain, then we can also talk of properties of “*a*”.

Given a domain, the terms of a language like FOL refer to the elements of the domain. Reification allows us to add more “elements” to the *universe of discourse* by extending the conceptual space to include symbolizations of higher order constructs like those representing events and actions.

Reification also allows us to represent compound objects as entities composed from smaller parts. And almost everything that we talk about is in fact a compound object. For example, at one of the lowest levels of detail, the hydrogen atom is a reified concept made up of a proton and an electron in perpetual entanglement. It does not exist independent of its constituents, but is made of them. Likewise the other elements like carbon, silicon and sulfur too are made of subatomic particles. According to the atomic *theories* of yesteryears, when atoms were supposed to be indivisible, the world is made up of a large collection of atoms existing in many clusters and interacting in myriad ways. *Conceptually* as we move up the level of

detail we *create* concepts at the molecular level, biological level, societal level, geological level and astronomical level with the same set of atoms. In that sense the world as we *know* it is our own creation. These reified concepts⁴ we create in our minds have obviously been critically instrumental in us being able to *comprehend* the world and interact *meaningfully* with it, including actions like reading these very words that you are doing right now.

Every discipline of science and art *creates* representations at a level of detail suited to it. An expert in that discipline is well versed in its ontology. In modern times no individual can be a *philosopher* in the sense of olden days. We exist as a society of heterogeneous minds, each specialized in some domain. Perhaps in this context it might be too much to expect a single computer (program) to be omniscient, and one must be willing to accept artificial intelligence if it too manifests itself in a heterogeneous form.

Meanwhile the task of devising an integrated knowledge representation that can be central to diverse domains and different forms of reasoning will surely keep AI researchers occupied for at least a decade.

The objective of writing computer programs demands that we create the conceptual world that is of interest in an explicit and formal manner, because computers at the lowest level are simply syntactic processors⁵. Intelligent or meaningful behavior can only emerge at a higher level. The task of creating representations for conceptual structures is to a large extent domain dependent, and is generally referred to as knowledge engineering. In this chapter and the next, we look at some general principles that can guide the choice of our predicates, and the reasoning patterns that we can adopt while building real systems.

We look at two streams of reasoning that reification allows us. One is the ability to reason about actions explicitly. We shall look at both a logical approach with the event calculus and an experiential approach with structured knowledge (partly in the next chapter). The second, dealt with in the next section, is the possibility of evolving a uniform representation scheme based on triples that will allow us to describe arbitrary relations.

13.3 RDF and the Semantic Web

Consider the following set of predicates and the English sentences they are to presumably be interpreted as

Hit(Azizi32, Anuun12)	: Azizi hit Anuun
Hit(Azizi32, Anuun12, Oct_12)	: Azizi hit Anuun on October 12
HitLoc(Azizi32, Anuun12, Paris)	: Azizi hit Anuun in Paris
HitInstr(Azizi32, Anuun12, Stick)	: Azizi hit Anuun with a Stick

One can construct other variations and combinations of such statements. Observe that the predicate “Hit” in the first and the second instances are different, because they are of different arity. Defining different predicates for the different combinations of things one wants to assert is not only cumbersome; it also increases the inference load on the reasoning system. One will need rules like,

$$\forall x \forall y \forall z (\text{Hit}(x, y, z) \supset \text{Hit}(x, y))$$

to be able to ask queries like “Who did Azizi hit?” assuming that the answer extracting routine is programmed to handle the consequent in the implication.

There are other issues with the above formalizations as well. For example is “Stick” a constant or a variable of a certain type? That is, is one talking of a specific object in the domain (mapped to by the constant “Stick”) or a generic one? If it is the latter, and the phrase “a stick” certainly suggests so, then *FOL* would require us to state the sentence as,

$$\exists x (\text{Stick}(x) \wedge \text{HitInstr}(\text{Azizi32}, \text{Anuun12}, x))$$

In the above sentence, *Stick* is now a predicate or category, and “*x*” is the variable that refers to an element of that category. If we were to use a typed version of *FOL* called *Many-Sorted FOL* then one could introduce a type or a category or a sort S_{stick} and use the sentence $\text{HitInstr}(\text{Azizi32}, \text{Anuun12}, \text{Stick})$ where *Stick* now refers to a variable from the sort S_{stick} . We will look at sorted logic in a little bit more detail when we look at the Event Calculus below. One may also notice that we have glossed over the treatment of *time* and used the term *Oct_12* to somehow stand for October 12.

Some of the issues can be tackled by introducing an *event type* or *sort* into the system, reify the hitting event, and describe its properties. We can also adopt a uniform description framework constituting of a triple made up of <subject, predicate, object> or <subject, property, value> in which each statement conforms to the triple syntax. Additional information about an event can then be expressed by asserting more statements (triples) instead of arbitrarily increasing the arity of the predicate. The hitting episode can now be describes as,

(Hitting_event Instance e_{31})	: e_{31} is a (type of) Hitting_event
(e_{31} Actor Azizi32)	: The actor of e_{31} is Azizi
(e_{31} Object Anuun12)	: The object of e_{31} is Anuun
(e_{31} Date Oct_12)	: The date of e_{31} is October 12
(e_{31} Loc Paris)	: The location of e_{31} is Paris
(e_{31} Instrument Stick)	: The Instrument of e_{31} is Stick

Here *Hitting_event* is a type of an event. The first sentence says that e_{31} is an instance of *Hitting_event*, and the rest of the sentences provide more information about this event. Observe that we have not stated explicitly that the hitting event has happened. The Event Calculus will address that in Section 13.5. Here we have assumed instead that the once we have an instance of the hitting act it means that the act actually happened. We can still add other statements that the event actually happened, or that the event was being planned, or even dreamt about.

In *FOL* syntax we have been using the same statement can be expressed as,

Instance(Hitting_event, e_{31})	: e_{31} is a (type of) Hitting_event
Actor(e_{31} , Azizi32)	: The actor of e_{31} is Azizi
Object(e_{31} , Anuun12)	: The object of e_{31} is Anuun
Date(e_{31} , Oct_12)	: The date of e_{31} is October 12
Loc(e_{31} , Paris)	: The location of e_{31} is Paris
Instrument(e_{31} , Stick)	: The Instrument of e_{31} is Stick

The fact that we have separated the different aspects of the event into independent statements means that the inference engine would have to search for every additional piece of information that is asked for. In the next chapter we will look at ways of alleviating the problem of search by establishing explicit connections between related elements.

Resource Description Framework (RDF) (Lassila and Swick, 1998) is a representation scheme that has evolved to describe the data available on the World Wide Web⁶. It is an XML application (i.e., its syntax is defined in Extensible Markup Language) customised for adding *metadata* in Web documents. Metadata simply means *data about data*. The basic idea is to annotate data available on the web with labels. Such labels, or metadata, can provide (semantic) details of the relations between entities (resources). An *RDF* statement is a triple,

<subject, property, value> or <subject, predicate, object>

where the subject is a *resource* that is being described, marked by the *RDF* word “about”, the property is a resource from some *namespace*, and the value is either a resource or a value from some *valuespace*. A resource is anything that can be uniquely identified.

In order to avoid the problem of ambiguity of terms, *RDF* uses the notion of a *Uniform Resource Identifier (URI)* that uniquely identifies a resource. The *URI*’s are used to create references in a standard form, usually in XML though other forms are possible⁷, which can be parsed by other programs or browsers. *RDF* employs a notion of a *namespace* to identify standard names for its elements. A namespace is used to provide uniquely named elements, and may be specified by “xmlns” (for xml namespace) with a value that is a *URI*. The following are examples of namespaces,

```
xmlns:xhtml = "http://www.w3.org/1999/xhtml#"
xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs = "http://www.w3.org/2000/01/rdf-schema#"
```

The entry on the left hand side following “xmlns:” is the short form that can be used instead of the longer right hand side. For example the second entry above states, for the benefit of any program accessing that resource, that the enclosing document is an *RDF* document, and that the *rdf:RDF* tag resides in this namespace. One can also create one’s own namespaces, as for example,

```
xmlns:composition = "http://www.muzicsite.zzzz/composition#">
```

The use of *URIs* is instrumental in extending the domain of the statements to the entire World Wide Web, and the process can be called *webizing* (Berners-Lee, 1998). An application on one machine can talk about the properties of a resource described on some other machine, thus creating a *semantic web* of resources. It must be kept in mind that the goal of the semantic web is to *share* information, and in particular to enable programs to share information in a meaningful manner across the web. It is a representation scheme, and does not address the issues of reasoning in itself.

Since *RDF* is designed to share information generated by diverse people one cannot expect the information available on the web to conform to a fixed schema. Anyone creating an information resource can add more data. For example a car manufacturer’s site may give information about a particular car (resource), while a group of car aficionados may add their opinions as properties and their values about the same resource (on their own site). To share information, the different sites may need to use a common vocabulary and provide secure *XML endpoints* to other users. One way to do this would be for different sites to use a same *base ontology* to define a vocabulary of terms and relations between terms. We discuss ontologies in more detail in Chapter 14.

Usually one collects all the properties about a particular resource together into

a collection as illustrated in Figure 13.1. The figure depicts how a typical site providing music may organize information about the musical items in its collection. All the properties and their values are collected together in one *Description*, though it is not necessary to do so. Obviously keeping such information together would be useful from a computational point of view, and we will explore this idea more in the next chapter.

The statements in Figure 13.1 are as follows. Line 1 declares the *XML* version being used. *XML* documents can be visualized as tree structures. Line 2 is called the *root element* of the document and says that everything up till the closing element `</rdf:RDF>` is part of the document. Observe that the element in line 2 extends to lines 3 and 4, after which the closing bracket `>` occurs. Line 3 says that the short form *“rdf”* maybe used to refer to the namespace *“http://www.w3.org/1999/02/22-rdf-syntax-ns#”*. An example of a word from this namespace is in line 5 which says that *“Description”* is a *rdf word*. Line 4 defines another namespace *“composition”* that contains the terms (album, artist, lyrics etc.) that are the predicates or properties used to describe the musical items (resources).

Each musical item stored in the directory *“title”* on the site is described within the `<rdf:Description> ...</rdf:Description>` tags. Line 6 says that the description in lines 7 to 13 is *about* the resource *“Dashte Tanhaf”*. Line 7 says that the composition comes from the *album “Masters sing Faiz”*, line 8 identifies the *artist* as *“Iqbal Band”*, and so on.

Lines 15 to 24 describe another musical piece *“Jamuna ke Teer”* and lines 25 to 35 describe one called *“Spanish Dance”*. Observe that the artist element in the last piece is a *sequence* of two ordered elements, marked by the *rdf* word *“Seq”*. Other similar *rdf* words are *“Bag”* for unordered collections, and *“Alt”* to specify a list of alternative values of which only one can be selected.

```

1. <?xml version = "1.0"?>

2. <rdf:RDF
3. xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4. xmlns:composition = "http://www.muzicsite.zzz /composition#">

5. <rdf:Description
6. rdf:about = "http://www.muzicsite.zzz/title/Dashte Tanhai">
7.     <composition:album> Masters sing Faiz </composition:album>
8.     <composition:artist> Iqbal Bano </composition:artist>
9.     <composition:musicalForm> Ghazal </composition:musicalForm>
10.    <composition:medium> Vocal </composition:medium>
11.    <composition:lyrics> Faiz Ahmed Faiz </composition:lyrics>
12.    <composition:recording> EM I </composition:recording>
13.    <composition:year> 2007 </composition:year>
14. </rdf:Description>

15. <rdf:Description
16. rdf:about = "http://www.muzicsite.zzz/title/Jamuna Ke Teer">
17.     <composition:album> Golden Raga Collection </composition:album>
18.     <composition:artist> Pandit Bhimsen Joshi </composition:artist>
19.     <composition:musicalForm> Thumri </composition:musicalForm>
20.     <composition:raag> Raga Bhairavi </composition:raag>
21.     <composition:medium> Vocal </composition:medium>
22.     <composition:recording> Times Music </composition:recording>
23.     <composition:year> 1997 </composition:year>
24. </rdf:Description>

25. <rdf:Description
26. rdf:about = "http://www.muzicsite.zzz/title/Spanish Dance">
27.     <composition:album> Carmen Fantasy </composition:album>
28.     <composition:artist>
29.         <rdf:Seq>
30.             <rdf:li>Sergei Nakariakov</rdf:li>
31.             <rdf:li>Alexander Markovich</rdf:li>
32.         </rdf:Seq>
33.     </composition:artist>
34.     <composition:musicalForm> Virtuoso </composition:musicalForm>
35.     <composition:medium> Trumpet and Piano </composition:medium>
36.     <composition:composer> Manuel de Falla </composition:composer>
37.     <composition:recording> TELDEC </composition:recording>
38.     <composition:year> 1994 </composition:year>
39. </rdf:Description>

40. .
41. .
42. </rdf:RDF>

```

FIGURE 13.1 An example of the use of *RDF* to provide metadata for music pieces. Italics only used to demarcate the content for the reader.

The figure below depicts part of the data in a graphical form. Each *RDF* triple is represented by an arc labeled with the predicate from the subject to the object. The values in the square boxes are *literals* and are to be taken literally (without any interpretation). We could have made *URI*'s for some of them, for example the artist and the album title. This approach of using *RDF* triples to store data leads to the idea of *graph databases*.

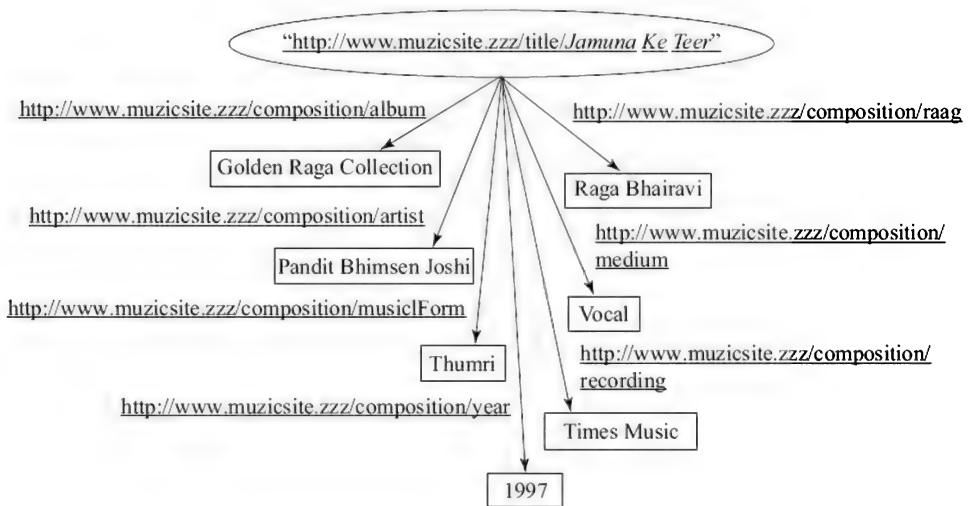


FIGURE 13.2 The underlying graph model for an *RDF* Description. The oval represents a resource, and the boxes represent values.

The key features of *RDF* are the use of namespaces and URIs to enable the sharing of descriptions of properties of resources. It does not allow one to define new categories or classes. For that we need an extension of *RDF* known as *RDF Schema* (RDFS), which allows user defined classes and the expression of sub-class properties. We will take that up in the next chapter in which we explore the representation and reasoning over taxonomies.

If we need to define new applications or domain specific classes then we need to use an extension of *RDF* that allows such class definitions. For example, we might want to define “*thumri*” as a (type of) class to which many compositions belong. We can then define “*hindustani*” as a type of class, and be able to make statements like *thumri* is a subclass of *hindustani*. A language that allows us to do so is the resource description framework schema (RDFS). In the next chapter we will also look at the Web Ontology Language (OWL) which is an extension of RDFS. Figure 13.3 shows an example of statements in RDFS.

The primary concern of *RDF* and *RDFS* is to express information in a uniform manner so that it can be easily processed by algorithms. An important feature is the use of a uniform vocabulary to avoid the problem of translations between different nomenclatures. We look at an example below.

13.3.1 The Dublin Core Metadata Initiative

One noteworthy effort in this direction is the Dublin Core Metadata Initiative which defines the metadata needed to describe documents on the web (DCMI, 1999) (also see <http://dublincore.org/>). The Dublin Core element set (dc) defines the basic vocabulary of a language used for describing resources/documents

```

<?xml version = "1.0"?>

<rdf:RDF
xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs = "http://www.w3.org/2000/01/rdf-schema#"
xml:base = "http://www.musicology.example/music#">

<rdfs:Class rdf:ID = "music" />

<rdfs:Class rdf:ID = "hindustani">
  <rdfs:subClassOf rdf:resource = "#music"/>
</rdfs:Class>

<rdfs:Class rdf:ID = "khayal">
  <rdfs:subClassOf rdf:resource = "#hindustani"/>
</rdfs:Class>

<rdfs:Class rdf:ID = "thumri">
  <rdfs:subClassOf rdf:resource = "#hindustani"/>
</rdfs:Class>

</rdf:RDF>

```

FIGURE 13.3 Class definitions in RDFS.

(Baker, 2000). The DCMI Abstract model of the resource is as follows. The abstract model of the *resources* described by *descriptions* is as follows:

- Each *described resource* is described using one or more *property-value pairs*.
- Each *property-value pair* is made up of one *property* and one *value*.
- Each *value* is a *resource*—the physical, digital or conceptual entity or *literal* that is associated with a *property* when a *property-value pair* is used to describe a *resource*. Therefore, each *value* is either a *literal value* or a *non-literal value*:
 - A *literal value* is a *value* which is a *literal*.
 - A *non-literal value* is a *value* which is a physical, digital or conceptual entity.
- A *literal* is an entity which uses a Unicode string as a lexical form, together with an optional language tag or datatype, to denote a *resource* (i.e. "literal" as defined by *RDF*).

The *Dublin Core* set constitutes of the following property names (see (DCES))—

- | | |
|-------------------|-----------------------------------|
| • dc: contributor | (contributors to the document) |
| • dc: coverage | (where the resource is located) |
| • dc: creator | (the creator) |
| • dc: date | (date of publishing) |
| • dc: description | (a brief account of the resource) |
| • dc: format | (format of presentation) |
| • dc: identifier | (a <i>URI</i> for the resource) |
| • dc: language | (language of content) |

- dc:publisher (publisher)
- dc: relation (a reference to a related resource)
- dc: rights (copyright)
- dc: source (origin on content)
- dc: subject (the topic described in the resource)
- dc:title (the title of the resource)
- dc:type (the nature or genre of the content)

These property names are available in the *dc* namespace, as indicated below.

```
<rdf:RDF xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dc = "http://purl.org/dc/elements/1.0/">
```

A larger vocabulary of *terms* aimed to qualify or refine the fifteen *elements* above is made up of the following resources⁸

abstract, accessRights, accrualMethod, accrualPeriodicity, accrualPolicy, alternative, audience, available, bibliographicCitation, conformsTo, contributor, coverage, created, creator, date, dateAccepted, dateCopyrighted, dateSubmitted, description, educationLevel, extent, format, hasFormat, hasPart, hasVersion, identifier, instructionalMethod, isFormatOf, isPartOf, isReferencedBy, isReplacedBy, isRequiredBy, issued, isVersionOf, language, license, mediator, medium, modified, provenance, publisher, references, relation, replaces, requires, rights, rightsHolder, source, spatial, subject, tableOfContents, temporal, title, type, valid

Each term is specified with the following minimal set of attributes:

Name:	A token appended to the <i>URI</i> of a DCMI namespace to create the <i>URI</i> of the term.
Label:	The human-readable label assigned to the term.
URI:	The Uniform Resource Identifier used to uniquely identify a term.
Definition:	A statement that represents the concept and essential nature of the term.
Type of Term:	The type of term as described in the DCMI Abstract Model

For example, the term *isReferencedBy* is described as,

Name:	isReferencedBy
URI:	http://purl.org/dc/terms/isReferencedBy
Label:	Is Referenced By
Definition:	A related resource that references, cites, or otherwise points to the described resource.
Type of Term:	Property
Refines:	http://purl.org/dc/elements/1.1/relation
Refines:	http://purl.org/dc/terms/relation
Version:	http://dublincore.org/usage/terms/history/#isReferencedBy-003

Note: This term is intended to be used with non-literal values as defined in the DCMI Abstract Model (<http://dublincore.org/documents/abstract-model/>).

In the next section we look at an endeavour to link many such data resources into a large graph database freely accessible over the world wide web.

13.3.2 The Linked Open Data Cloud

The *LOD* (Linking Open Data) project⁹ is espoused by the erstwhile SWEOW (Semantic Web Education and Outreach) Interest Group¹⁰ of the W3C consortium. The goal of the project is to make data freely available to everyone on the web. The *LOD* cloud refers to a network of servers that host data in the form of *RDF* triples from various domains. Thus the *LOD* cloud can be seen as a massive graph database on the web.

The goal is to make the various data sets available on the web in a well defined format under the Creative Commons license¹¹, and move from a *web of documents* towards the *web of data* also referred to as the *Semantic Web* or *Web 3.0*. It would then be possible for programs to search for data over the web and use the results for further processing directly. The different data sources are also connected by *RDF* links to enable programs to navigate between related data sets. Development of Linked Data browsers and Linked Data search engines facilitate the retrieval process (Bizer et al, 2009). Thus a program has access to the *entire* data from the different sources and can use it to compute answers to queries.

Tim Berners-Lee prescribed the following “Linked Data principles” for people publishing data on the web (Berners-Lee, 2006).

1. Use *URIs* as names for things
2. Use *HTTP URIs* so that people can look up those names
3. When someone looks up a *URI*, provide useful information, using the standards (*RDF*, *SPARQL*)
4. Include links to other *URIs*, so that they can discover more things

The *URIs* are used to uniquely identify objects and the *HTTP* protocol is used to access them. The data and the links are themselves described using *RDFS* and *OWL* (see also Chapter 14).

Figure 13.4 depicts the status of the *LOD* cloud in 2007 (Cyganiak, 2007). Each node in the cloud represents a distinct data set. Each arc represents an *RDF* link between elements of the two datasets. The arcs are directed indicating the direction of links and the thickness of the edges is an indication of the number of links between the two nodes (Linked Data sets).

The content of the cloud is diverse in nature, comprising data about geographic locations, people, companies, books, scientific publications, films, music, television and radio programmes, genes, proteins, drugs and clinical trials, online communities, statistical data, census results, and reviews (Bizer et al, 2009). For example, the *Friend Of A Friend* (FOAF) project¹² is a .*RDF* based machine readable dataset describing people. And *DBLP Berlin* is a *RDF* version of the *DBLP* database of University of Trier which provides bibliographic information on major computer science journals and conference proceedings, maintained by the University of Berlin. Likewise *DBpedia* is a data set that is composed by extracting structured content from Wikipedia. It allows users to query properties and relationships associated with Wikipedia resources, including links to other related

datasets. And *openCYC* is an ontology of everyday common terms.

A catalog of the projects that are linked in the *LOD* cloud is maintained by Richard Cyganiak and Anja Jentzsch¹³.

Collectively, the 295 data sets consist of over 31 billion *RDF* triples, which are interlinked by around 504 million *RDF* links as of September 2011⁸.

The Figure 13.5 (see the source (Cyganiak and Jentzsch, 2011) for an enlarged readable view) shows that the number of datasets in the *LOD* cloud has been steadily increasing¹⁴.

The Figure 13.6 below shows another view of a part of the above graph in a little more detail.

13.3.3 Querying Graph Databases with SPARQL

RDF is a directed, labeled graph data format for representing information in the Web. *RDF* data stores can also be queried using their own query language —*SPARQL* (*SPARQL* Protocol And *RDF* Query Language)¹⁵. A query may be placed at the *SPARQL* endpoint of any *RDF* database.

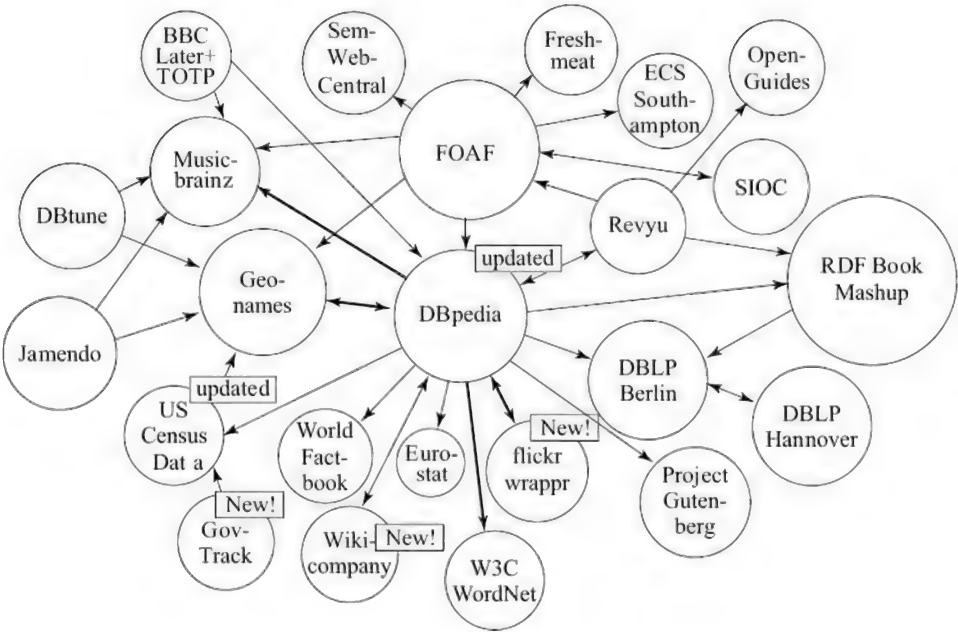


FIGURE 13.4 The Linking Open Data cloud in 2007 (Cyganiak, 2007).

Like its relational cousin *SQL*, *SPARQL* also employs a *SELECT* statement to determine which subset of the selected data is returned. It also uses a *WHERE* clause to define *graph patterns* to find a match for in the query data set. A graph pattern in a *WHERE* clause consists of the subject, predicate and object triple to find a match for in the data. The *FROM* clause is used to identify the dataset from where the answers are to be retrieved, and clauses like *ORDER BY* and *DISTINCT* can be used to control the manner in which the results are presented. Observe that the results will be in a machine readable *XML* document. The *PREFIX* keyword allows us to define prefix labels for *IRIs* (Internationalized Resource Identifiers). An *IRI* extends the way *URIs* are defined.

We illustrate a *SPARQL* query with the following example. The query is to

retrieve all names of *ghazal* albums and their artists in 2012. The *PREFIX* statement says that the properties names prefixed by “composition:” are defined in the corresponding namespace. The *SELECT DISTINCT* clause says that we want distinct values for the variables *?name* and *?album*. Variables names in query patterns are prefixed with “?”.

```
PREFIX composition: <http://www.muzicsite.zzz /composition#>
SELECT DISTINCT ?name ?album
FROM <http://www.muzicsite.zzz /title>
WHERE {
    ?title composition:artist ?name
    ?title composition:musicalForm Ghazal
    ?title composition:album ?album
    ?title composition:year 2012
ORDER BY ?name
```

The graph patterns are defined in the *WHERE* clause. The first pattern extracts the name of the artist for the resource and binds it to the variable *?name*. The second one imposes a constraint that the musical form must be “Ghazal”. The third one extracts the name of the album in the variable *?album*, and the fourth one restricts the year to the value 2012. The *ORDER* clause says that the output, which is the name of the artist and the album, should be sorted on the *?name* variable.

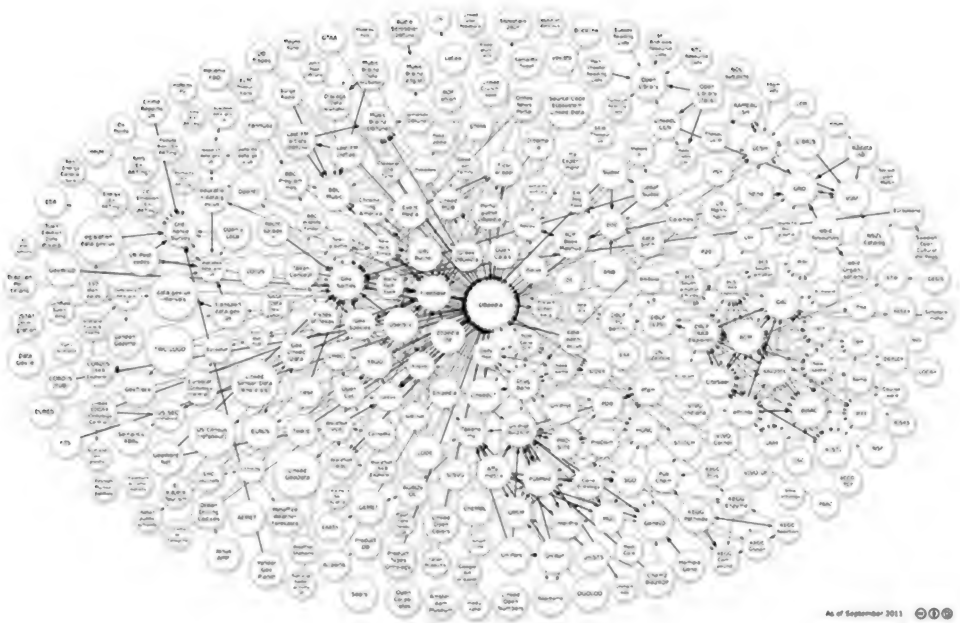


FIGURE 13.5 The Linking Open Data cloud in 2011 (Cyganiak and Jentzsch, 2011)

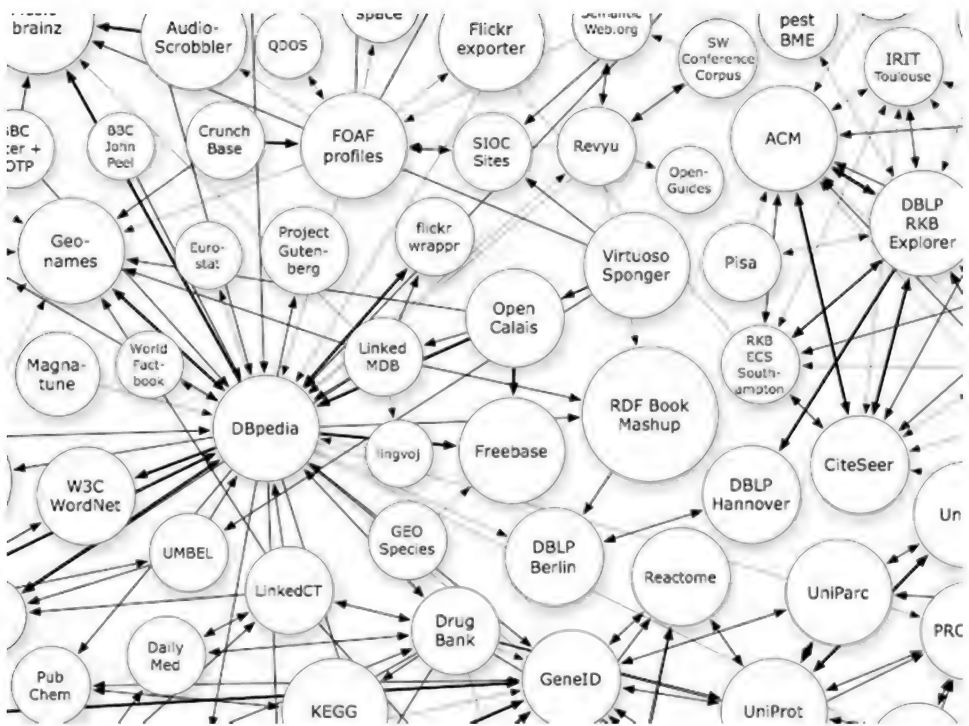


FIGURE 13.6 Apart of the *LOD* cloud of Figure 13.5 (from <http://linkeddata.org/>).

Apart from the *SELECT* form of the query, *SPARQL* allows three more forms of querying the database. The *CONSTRUCT* query returns a single *RDF* graph formed by taking each query solution in the solution sequence, substituting for the variables in the specified graph template, and combining the triples into a single *RDF* graph by set union. The *ASK* form returns a *yes* or *no* answer to whether the query pattern has a solution. The *DESCRIBE* form returns a single result *RDF* graph containing *RDF* data *about* resources. The query pattern is used to create a result set, from which a description is extracted and assembled into a single *RDF* graph. The interested reader is directed to (Prud'hommeaux and Seaborne, 2008) for details.

13.4 Properties

Each *term* of the *FOL* is mapped to an object in the domain. We have also introduced event and actions, and added them as reified objects to the domain. We now look at how certain properties of objects, and events, may be described.

We would like to express the following kinds of statements, with emphasis on the words in *italics*.

- The ball is *red*.
- The ball is *heavy*.
- The pencil is *longer than* the pen.
- The tree is *five feet tall*.
- Subun ate up the chocolate *quickly*.
- It took the train *three hours* to reach Mysore.

We *could* of course simply translate the statements directly with the adjectives and the verbs forming the predicates as follows.

- Red(Ball22)
- Heavy(Ball43)
- LongerThan(Pencil32, Pen67)
- Height(Tree345, 5_feet)
- Eat(Subun11, Chocolate23, Quickly, Past_perfect)
- Duration(Train_journey56, Mysore34, 3_hours, Past_perfect)

But this would be rather *ad hoc*, and not conducive to building and sharing large amounts of knowledge. In addition there is a problem trying to define the semantics of the above sentences, since the predicates have all kinds of arguments.

Properties like colour, weight, height and length can be expressed in various ways. The choice of adjectives as predicates raises the question of what the predicates mean. We have seen earlier that predicates like *Man* and *Human* can be used to define categories of objects. Thus, *Man*(*x*) stands for the set of objects that are instances of men¹⁶. Does the predicate *Red*(*x*) mean something similar? A similar question might be raised about the predicates *Mortal*, and *Bright* we used earlier.

Obviously the relation between the concept of (being) a Man and the individual Socrates, is not quite the same as the relation between our concept of being red and the individual ball. The English language distinguishes between the two relations by employing different grammatical structures, “The teacher is a man” *vis a vis* “The ball is red”. The presence of the article “a” identifies “man” as a noun, while its absence identifies “red” as an adjective. We see red as a *property* of the ball, while we see Socrates as being an *instance* of a *class* called *Man*. Is *Red* a class too? In a way we could say that our notion of redness comes from all the red coloured objects we have seen, but we cannot say that the ball is an instance of “red”. Another possibility is to code it, like one might in *RDF*, as,

Colour(Ball22, red)

Now we are saying “Colour” is a property (or predicate) that describes the subject “Ball22” and its value is a literal named “red”. This is a little better since we do tend to think of colour as a property of a (physical) object, and the statement reflects that relation. In the above statement we have modeled “red” as an (reified) object in the domain. We could have also used *Colour* as a function to assert,

Colour(Ball22) = red

We could even relate it to other colours by functions like,

Similarity(red, orange) = 0.78

We could also define a category called *Colour_class* of which *red* is an instance, expressed by the statement *Colour_class*(*red*) or *Instance*(*Colour_class*, *red*) being *true*.

Weight and length are properties we tend to use both quantitatively (5 kg, 3 km) and with fuzzy linguistic descriptors (light, short). Quantitative descriptions themselves tend to employ different measures, and one needs to find mechanisms to reason with different measures, for example, 26 miles 385 yards = 42.195 km.

Let us say we want to talk about the length of an object *Pencil41*. Now if we can think of a reified object that represents, or that *is*, the length of *Pencil41*, we could point to that object of *type length* by using a function called “length” as *length*(*Pencil41*). Let us say that the pencil is five inches long. We could think of another function “inches” that takes an argument of the *type length* and returns a

number, in this case 5 (as described in (Brachman and Levesque, 2004)). Likewise we could define a function “cms” that returns the value 12.7 in centimetres. That is,

inches(length(Pencil41)) = 5
cms(length(Pencil41)) = 12.7

We could also write conversion equations like,

inches(length(Object)) * 2.54 = cms(length(Object))

We could now define the notion of *longer than* by defining an ordering on the objects of type length as follows.

$(\text{length}(O_1) > \text{length}(O_2)) \equiv (\text{cms}(\text{length}(O_1)) > \text{cms}(\text{length}(O_2)))$

assuming we have an ordering on numbers.

But now what *is* a number? (McCulloch, 1961)¹⁷ At the most basic level, numbers have obviously something to do with counting. We say that two sets have the *same number* of elements if we can establish a bijection between them. Bertrand Russell said that the number is “a *class of all those classes that can be put in one to one correspondence with it*”. A number is certainly an abstract concept, and is a perfect example of a reified entity. The *numeral* 7, or VII, or the word “seven”, for example corresponds to the reified entity that is the *number* 7, which is an element of a set (of natural numbers) that can be defined inductively. The number 0 maps to the cardinality of the empty set. The following definition of the set of numbers is due to John von Neumann. Each number corresponds to a set, which contains that many elements. Zero corresponds to the empty set. We say zero is equal to the empty set. Given a number N , the next number $s(N)$ is the union of the set containing N and all the members of (the set) N . That is,

$s(N) = \{N\} \cup N$ (remember N is a set)

For readability, we use the notation Φ to stand for the empty set.

$0 = \Phi$

$1 = \{\underline{0}\} = \{\underline{\Phi}\}$ **note 0 or Φ has no elements**

$2 = \{\underline{1}, 0\} = \{\{\underline{\Phi}\}, \Phi\}$

$3 = \{\underline{2}, 1, 0\} = \{\{\{\underline{\Phi}\}, \Phi\}, \{\Phi\}, \Phi\}$

$4 = \{\underline{3}, 2, 1, 0\} = \{\{\{\{\underline{\Phi}\}, \Phi\}, \{\Phi\}, \Phi\}, \{\{\Phi\}, \Phi\}, \{\Phi\}, \Phi\}$

.

.

Thus a number is greater than all its predecessors, and the successor of a number is made by taking the elements of the number and adding one more element, the number itself, shown underlined above.

Given the above definition of the successor function we can define the set of natural numbers as successive successors of the number 0.

The set of numerals in the language $\{0, 1, 2, 3, \dots\}$ can then be mapped to the reified objects $\{0, s(0), s(s(0)), s(s(s(0))), \dots\}$. Note that we have overloaded the symbol “0” to stand both for the numeral 0 and the number 0.

Another way we could talk about length is to deal with the reified object length(Object) directly (Russel and Norvig, 2009). The function inches would then take a number as input and return an appropriate object of type Length. Then we could say,

length(Pencil41) = inches(5) = cms(12.7)

This has the advantage that we can directly talk of reified objects like “5 inches” as length that in inches measures 5 units, and define predicates like,

height(Aditi96, inches(66), year(2008))

One would also find it easier to represent statements like “It was a three hour long odyssey”.

Conversion now will need multiplication inside the brackets.

inches(X) = cms(X * 2.54)

One thing we might want to do with measured quantities is to add them, and to compare them. In the first notation one can assert that only same unit measures can be compared. Then since the functions return numbers we can simply make statements like,

km(distance(Delhi, Chennai)) > km(distance(Delhi, Mumbai))

This is a bit unnatural, since we want to say that Chennai is farther from Delhi than Mumbai is, and we do not want to talk about units of measurement unnecessarily. We can get around this by asserting the following equivalence,

(distance(Delhi, Chennai) > distance(Delhi, Mumbai)) \equiv
(km(distance(Delhi, Chennai)) > km(distance(Delhi, Mumbai)))

The second notation returns objects of the abstract length type, over which we would have to define the ordering as dependent upon the underlying order of numbers, for example as follows.

(km(X) > km(Y)) \equiv (X > Y)

This says, for example, that if 2100 is greater than 1200, then 2100 km is “greater” than 1200 km, and vice versa.

We may also need to define the addition function to reason with situations where quantities need to be added. For example “Shyam ran for 3 km and walked another 2 km. How much distance did he cover?”

In the first notation, addition is defined naturally, and one will assert the facts as,

km(length(firstLeg))	= 3
km(length(secondLeg))	= 2
km(length(total))	= km(length(firstLeg)) + km(length(secondLeg))
	= 3+2
	= 5

In the second notation, we would assert the facts as,

length(firstLeg)	= km(3)
length(secondLeg)	= km(2)
length(total)	= length(firstLeg) + length(secondLeg)
	= km(3) + km(2)
	= km(3 + 2)
	= km(5)

This needs an extra inference step based on the rule

$$\forall X \forall Y (km(X) + km(Y) = km(X+Y))$$

where X and Y have to be of type (or sort) number.

13.4.1 Fuzzy and Qualitative Categories

Humans tend to use linguistic terms to describe many categories with sets of values that objects may have, rather than use actual numerical values. This is especially useful if one needs to reason at an abstract level with a certain subsets of values. This may be used to define rules or for descriptions of behaviour or properties. Examples of such sentences, with the categories italicized, are,

- Rinse the clothes in *warm* water.
- The weather in Chennai is usually *hot*.
- *Tall* people make *good* basketball players.
- Sherpas in the Himalayas can carry *heavy* loads with *ease*.
- Taxi drivers do not like *short* rides.

It is not always possible to define crisp categories for properties that represent denote subsets of numeric values. How does one define the concept of warm water? One cannot say that the water is warm if the temperature is within some range $[t_{low} - t_{high}]$. This would imply that a temperature just below t_{low} is categorized as not warm while t_{low} itself is categorized as warm. Similarly, for the other categories, *hot*, *tall*, *heavy* and *short*. The notion of fuzzy sets introduced by Lotfi Zadeh in 1965 allows us to define meaningful semantics of such terms (Zadeh, 1965). A fuzzy set is a set which does not define membership of a set in crisp yes/no terms. Rather there are degrees of membership with which an element can belong to a set. While a *crisp* Set A is defined by its members, or by a characteristic function $c: A \rightarrow \{0, 1\}$, a fuzzy set A is defined by a membership function $m: A \rightarrow [0, 1]$. An element x can belong to a crisp set only if $c(x) = 1$, and it does not belong to it if $c(x) = 0$. On the other hand an element x can belong to a fuzzy set with a degree $m(x)$ which may be a value anywhere between 0 and 1. Thus if $m(x) = 0.7$ then we then we would be justified in saying that x belongs to A to a large extent.

Consider a rule in a fuzzy controller that says “*turn off the heater if the water is hot*”. Let us say the water is at 60 degree Celsius and we have a fuzzy membership function of hot that gives us a value of 0.8. How can we interpret or operationalize such a rule? One way would be to introduce randomized moves (like in simulated annealing) and have a well defined action that is *applied* with a probability proportional to the fuzzy membership value. The other, that is often used, is to determine whether a given temperature value belongs most to the fuzzy set *hot*, as compared to other sets like *warm* or *cool*, and then take the action specified in the fuzzy rule.

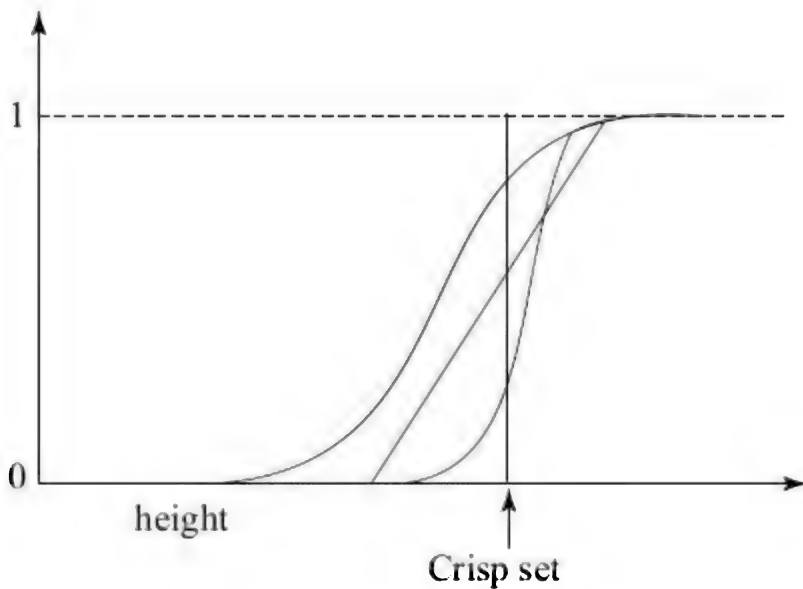


FIGURE 13.7 Different possible fuzzy membership functions for the notion of *tall*.

Defining fuzzy membership functions is the key feature in such applications. The shape of the membership function would determine the semantics of the fuzzy category. Figure 13.7 shows a set of possible fuzzy membership functions defining the notion of *tall* people. All the functions are monotonically increasing, which means that they all associate increasing height with increasing membership values. The values range between zero (definitely not tall) to one (definitely tall). Often to simplify computation one tends to use a linear function. The vertical line indicates a crisp set, with all values of height beyond the marked value being labeled tall. The others all agree beyond a point, but have different membership values around the crisp boundary. Observe that for a conservative application, for example recruitment of pilots, the crisp boundary may be shifted to the right.

In the examples given above, we have also marked the words *good* (basketball players) and (with) *ease*. While these are not linguistic terms for numerical values, they are nevertheless fuzzy categories in the sense that people would form fuzzy sets associated with them. One could think of numerical values which in principle indicate the proficiency in the game (like for example the Elo ratings in chess) or the amount of effort (the Sherpas have to make).

Fuzzy sets can be thought of as sets that are defined around some prototypical or idea values, and allow partial membership to values near the prototypical ones. For example, one may have in mind an interval of heights that defines “medium height” for a person, but people with heights outside but close to these values would still qualify to belong to the set, though to a lesser extent. Sometimes the intention of using categories is not based on linguistic goals, but to define sets of values that are equivalent for some form of reasoning. One is not interested in defining a function that captures how close the value is to the ideal value, but rather whether it belongs to an interval or not. The interval then defines a crisp set of values, but the values are such that they may be treated equivalently. This may happen if one wants to model a process without having access to numerical data. For example, the ice on a lake may be considered to be safe for skating after a certain thickness. Look at the following statements,

- The fluid in the brake pipes is *frozen*.

- Mohammed Ali was the *heavyweight* champion.

The symbols frozen and heavyweight may now be treated as values of variables. Variables taking such qualitative values are called as qualitative variables. Each such value in fact represents a set of values, which have the same effect of the reasoning process. We will look at *qualitative reasoning* later in Chapter 17.

13.4.2 Beyond Truth Functional Semantics

Given a language, what is its semantics? We have discussed the semantics of sentences from a logic perspective. The terms of a language *denote* objects in a domain. The meaning of predicates in *FOL* derives from the mapping to relations in the domain, and the truth values determine whether the sentence is *true* or not.

It has been argued that the use of language incorporates more than just assertion of facts. It is also concerned with making *useful* statements (see (Brandon, 1976), (Parikh, 1994), (Parikh and Ramanujam, 2003)). This is imperative when the different users of language may not necessarily completely agree on the truth values of assertions, but can still derive utility from communication.

In fact, Parikh defines some predicates as vague predicates for which logics and semantics cannot be defined. Consider the notion of colour. In a paradox mirroring the Sorites paradox, suppose that there are a *large* number (N) of colour patches labelled P_1 to P_N , and it is known that P_1 is red, and that P_N is not red. Further any two consecutive patches P_j and P_{j+1} are *visually* indistinguishable but *minutely* different in colour. This would mean that we would be inclined to use the same colour name for them. Then at which point traversing from 1 to N would we say that the patch is *not* red?

When one says “the woman in the red shirt” what does one mean? Does one talk of some truth functional semantics that says that the predicate *red(shirt)* is *true* based on some colour chart? Rohit Parikh has argued (Parikh, 1994; 2001) that one should work towards a *utility based semantics*. If I were to describe to you a person as the one wearing the red shirt, and if you are able to identify her (correctly), then my description of her wearing a red shirt is *useful* even if she is wearing what a purist might call a maroon shirt. This could happen for instance if the rest of the people in the group were all wearing differently shades of blue.

Utility thus plays an important role in what we say being in the context of some intention or purpose. The colours of lights on a traffic signal are typically red, amber and green. This is a utilitarian choice because the possibility of giving confusing signals is minimized. Ignore the placement of the three lights, or assume that you are speeding from far trying to decide whether you can cross the signal or not in time, and all you can notice is the colour of the light. Imagine what would happen if the red, amber and green were replaced by forest green, lime green and pigment green. Obviously there would be many more accidents. In fact there is utility even in choosing red for stopping, instead of green, because red light is of a longer wavelength and can be more clearly seen from far¹⁸.

13.5 Event Calculus

An intelligent agent needs to reason about actions and their consequences in a *changing* world. The world itself is described in terms of relations between objects expressed as *FOL* predicates. In a dynamic world however the value of the

predicate can change over time. We call such predicates *fluents*.

If we want to reason *about* the effects of actions on the world then we need to be able to treat actions and fluents as *arguments* to predicates that represent relations between them. Strictly speaking, this violates the definition of *FOL*, because arguments to predicates in *FOL* can *only* be terms, and terms are mapped to elements in the domain. However, we can circumvent this problem by *extending* the domain to include instances of actions and fluents. That is, for the purpose of reasoning with them, we add the *symbolic representations* of predicates and actions to the domain or the universe of discourse. We say that we have *reified* the actions and predicates.

We also need to introduce a representation of time, because we need to talk about *when* the actions happened and *what* fluent is *true* at a *given* time. The (classical) *FOL* operates in a 'mathematical' domain where predicates are either *always true* or *always false*, and in some sense time does not need to exist. The well known *Event Calculus* (EC) (Kowalski and Sergot, 1986), (Shanahan, 1999), (Mueller, 2006) introduces three (new) sorts – *events* or *actions*, *fluents*, and *time*. We can visualize the domain of event calculus as shown in Figure 13.8.

The subject matter for the Event Calculus comes from the *sorts* of *time*, *actions*, and *fluents*. The fluents themselves are predicates over what we can call the physical domain. The domain for the *EC* is the extended domain constituting of the time, actions and fluents domains.

The variables and constants of the *EC* belong to one of the following sorts.

An Event Sort, with variables {*e*, *e*₁, *e*₂ ...} The constants of the event sort will be the actions and exogenous events in the specific domain, like *unstack(block2, block6)*, *walk(home22, ramesh23, office34)*, *wakeUp(kumbhakaran1)*, *cyclone(nisha, 2008)*.

A Fluent Sort, with variables {*f*, *f*₁, *f*₂ ...} The constants of the fluent sort are the predicates from the domain, like *holding(block2)*, *loc(ramesh23, office34)*, *awake(kumbhakaran1)*.

A Timepoint Sort with variables {*t*, *t*₁, *t*₂, ...} The constants of the timepoint sort are numbers representing time points. In the Continuous Event Calculus (CEC) they may be real numbers, and in Discrete Event Calculus (DEC) they are integers.

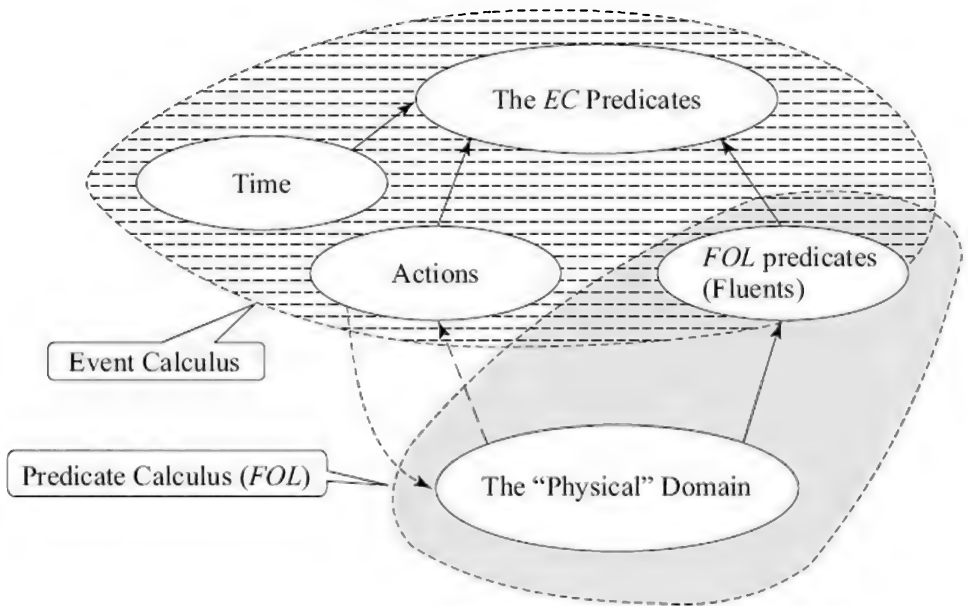


FIGURE 13.8 The domain for the *Event Calculus* is Time, Actions and Fluents.

The predicates of *EC* are relations between the different variables from the three sorts. Typically the relations describe the happening of events, and the relations between events, fluents and time. The commonly used predicates are described below.

Happens(e, t_1, t_2) Event e starts at t_1 and ends at t_2 . Observe that the event has a duration. For example, $Happens(Eclipse321, t_5, t_7)$ says that a particular eclipse happened between time points t_5 and t_7 . An instantaneous version of Happens can be defined as, $Happens(e, t) \stackrel{\text{def}}{=} Happens(e, t, t)$

HoldsAt(f, t) Fluent f is *true* at time point t . For example, $HoldsAt(Form(Glacier17, Solid), t_1)$ says that at time point t_1 *Glacier17* is in solid form. One may also define a predicate *Initially(f)* to assert that fluent f is *true* initially.

$$Initially(f) \stackrel{\text{def}}{=} HoldsAt(f, t_0)$$

Initiates(e, f, t) Event e occurs at time t and results in the fluent f becoming *true* after t . One might for example assert that the event of waking up initiates the fluent of being awake to be *true*, by $Initiates(wakeup(kumbhakaran1), awake(kumbhakaran1), t_3)$. In the DEC, it means that the fluent f is *true* at time $(t + 1)$ and later.

Terminates(e, f, t) Event e occurs at time t and results in the fluent f becoming *false* after t . In the DEC it means that the fluent f is *false* at time $(t + 1)$ and later. For the durative version of the action one can define the fluent to become *true* or *false* at either endpoint. For example, if the event is $Walk(Home, Actor, Office)$ from home to office then, the fluent $AtHome(Actor)$ becomes *false* (is terminated) at the start of the walk event, while the fluent $AtOffice(Actor)$ becomes *true* (is initiated) at the end.

Released At(f, t) The fluent f is released from the *commonsense law of inertia* at

time t . The commonsense law of inertia states that a fluent's truth value will not change unless affected - initiated or terminated—by an event. If a fluent is *released* from the commonsense law then it can fluctuate, and one cannot deduce its state. Releasing a fluent from the commonsense law is a mechanism to deal with certain kinds of uncertainty. For example, if you toss a coin then you release the fluent *Heads(Coin)* from the commonsense law, and it could take any value.

Releases(e, f, t) Event e occurs at time t , after which the fluent f is released from the commonsense law of inertia.

Trajectory(f_1, t_1, f_2, t_2) If fluent f_1 is initiated by an event that occurs at time t_1 then fluent f_2 will be *true* at time $(t_1 + t_2)$. This allows one to capture a causal relation between two fluents. For example, if *On(Stove)* is initiated by *Light(Stove)* at t_1 then the fluent *Temp(Water, SomeIncreasingFn(t_2))* is *true* at time $(t_1 + t_2)$.

AntiTrajectory(f_1, t_1, f_2, t_2) If fluent f_1 is terminated by an event that occurs at time t_1 then fluent f_2 will be *true* at time $(t_1 + t_2)$.

With the introduction of time there arises the problem of determining the truth value of fluents over *different instances* of time. What is *true* at one moment may be *false* at another time.

The following definitions are short forms for the equivalent formulas given after their descriptions.

Clipped(t_1, f, t_2) A fluent f that was *true* is made *false* sometime after or at time point t_1 and before t_2 . This is equivalent to the longer formula,

$$\exists e, t (Happens(e, t) \wedge (t_1 \leq t < t_2) \wedge Terminates(e, f, t))$$

Declipped(t_1, f, t_2) A fluent f that was *false* is made *true* sometime after or at time point t_1 and before t_2 . This is equivalent to the longer formula,

$$\exists e, t (Happens(e, t) \wedge (t_1 \leq t < t_2) \wedge Initiates(e, f, t))$$

PersistsBetween(t_1, f, t_2) The fluent f is not released from the commonsense law of inertia after time point t_x and up to and including time point t_2 . That is, it retains its truth value during the interval. This is a short form for,

$$\neg \exists t (ReleasedAt(f, t) \wedge (t_1 < t \leq t_2))$$

13.5.1 Effect of Events on Fluents

When an event happens that initiates a fluent, then the fluent becomes *true*. We make a simplifying assumption in the formulas below that the effect on the fluent is felt at the very moment the event happens. If one were to talk about the fluent being *true* at a later point one would need to add that the fluent was not *Clipped* in the interim period.

$$EC_1 (Happens(e, t) \wedge Initiates(e, f, t)) \supset HoldsAt(f, t)$$

Likewise if an event happens that terminates a fluent the fluent ceases to hold when it happens.

$$EC_2 (Happens(e, t) \wedge Terminates(e, f, t)) \supset \neg HoldsAt(f, t)$$

Events may release a fluent from the commonsense law of inertia, or they may terminate their released status.

$$EC_3 (Happens(e, t) \wedge Releasesfe, f, t)) \supset ReleasedAt(f, t)$$

$$EC_4 (Happens(e, t) \wedge (Initiates(e, f, t) \vee Terminates(e, f, t)) \supset \neg ReleasedAt(f, t)$$

Strictly speaking of course the effect should be felt *after* the event has happened. But between the instant the event happens and the instant when we want to evaluate the fluent nothing else should have happened. Hence, if we want to model a delayed response, we will need to include additional conditions in the left hand side along the lines of the ones in the inertia axiom below. For a detailed description the reader is referred (Mueller, 2006).

The Inertia Axiom

Using the above definitions we can infer that the value of a fluent remains the same if it remains under the commonsense law of inertia *and* is not clipped by some event.

$$IA: \{HoldsAt(f, t_1) \wedge (t_1 < t_2) \wedge PersistsBetween(t_1, f, t_2) \wedge \neg Clipped(t_1, f, t_2)\} \supset HoldsAt(f, t_2)$$

13.5.2 Reasoning

Different kinds of reasoning can be done with actions, events and states in a changing world (Shanahan, 1999).

Prediction Given an initial state and a narrative of events or actions, to deduce the fluents that are *true* in the final state. In the context of planning this is also known as projection.

Postdiction Given a final state and a series of moves that resulted in it, the task is to deduce the initial state.

Abduction Given the initial state and the final state, to find out the events that would transform the initial state into the final state. This is also planning, the task of finding the actions that transform the given state into the final one. Diagnosis may also involve finding events that resulted in something going wrong. Note that more than one solution may exist.

One may be called upon to do a combination of the above tasks. The task of a detective investigating a crime is to find out both what was *true* in the intermediate past, and what happened after that. A simpler version of such deductive reasoning has been described as *retrograde analysis* by Raymond Smullyan in his delightful collection of chess mysteries (Smullyan, 1979; 1992). A simple example of such a problem is given below. The book itself contains more challenging ones.

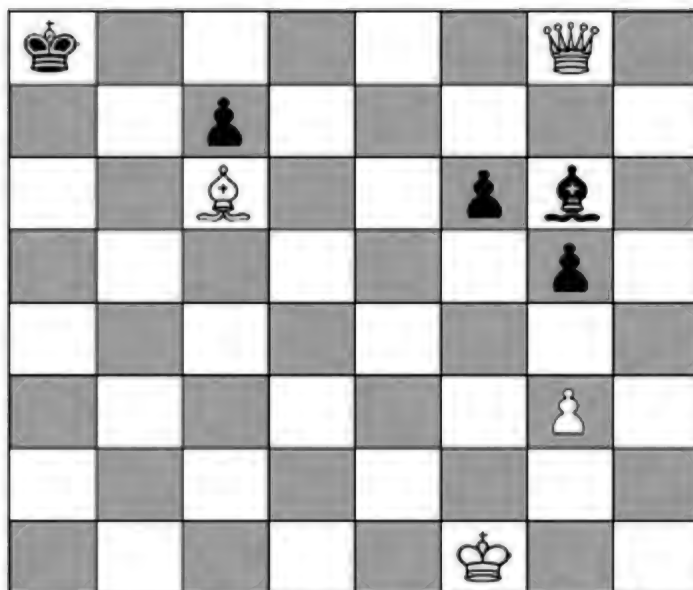


FIGURE 13.9 A simple problem of retrograde analysis.

The above figure represents the state of the board in a game of chess. Simply by looking at the board position, one can answer the question “Who played last, and what was the last move?” because in this situation only one move is consistent with the rules of chess. Notice that the black king is in check from both the white bishop and the white queen. Hence white must have moved last to administer the check. Further, before white’s move, the black king must not have been in check by either of them, since it would have had to take some defensive action and would then no longer be in check. This implies that both the checks must have been administered in white’s last move, one of them a discovered check, and that move could only have been moving the white bishop from e8 to c6¹⁹. The interested reader should look at the books by Smullyan for more intricate problems. A retrograde analysis problem requiring the solver to find the last 96 moves is attributed to the Hungarian chess player Gyula Breyer (see <http://mathpuzzle.com/retrograde.html>). A collection of problems and references can be found at (TRAC).

While retrograde analysis is an interesting example of logical (deductive) reasoning, the representation issues themselves are fairly straightforward. The rules of the game generate a state space in which actions link states. One does not really need to reason at a fine grained level where fluents might be required. Event calculus, on the other hand, equips us with machinery to reason at a more fine grained level. For example, one can infer that if a player has picked up a knight from the chessboard, she is holding the knight, and that the knight is no longer on the board. Such knowledge, while it does mirror facts, is not useful for the task of solving retrograde analysis chess problems. One could use it to infer for example that if the player has made a move then she must have picked up the piece and must have been holding it, and so on, provided that the context is that of playing on a physical chessboard. Such inferences, though interesting from the commonsense reasoning in AI perspective, and perhaps useful in solving a murder intrigue in which pawns from a chess set are laced with poison, are not likely to enamour the chess enthusiast.

The possibility that fluents *can* change value, can however lead to considerable

problems for *deduction* in an *open* world. In particular deductive reasoning is compelled to *assume* that all change that is happening is the one that is described explicitly or that logically follows from what is explicitly described. We shall look at the Event Calculus with an example that requires such an assumption.

13.5.3 A Bicycle Story

Consider the following narrative. “Nikhil filled up air in his bicycle. He intends to go to the restaurant at 9 a.m., eat his breakfast there at 9.30, and go to cricket practice at 10 a.m.”.

We assume that people do what they intend to if they can. This is the sense of intend that is used in the *Belief Desire Intention* model of agency (see Chapter 11).

Can we then infer that Nikhil is at cricket practice at 11 a.m.?

Let us state the facts in Event Calculus. We use time points t_1, t_2, t_3 and so on, and ordering relations between them. We use lower case words for constants and upper case for variables. For the sake of simplicity we assume actions are instantaneous.

Story N_1

1. HoldsAt(loc(nikhil53, home86), t_0)
2. Happens(fillAir(nikhil53, bicycle39), t_1)
3. Happens(planToRide(nikhil53, bicycle39, home86, restaurant66), t_3)
4. Happens(planToEat(nikhil53, restaurant66), t_5)
5. Happens(planToRide(nikhil53, bicycle39, restaurant66, cricketPractice11), t_7)

The sequence of predicates could (logically) be stated in any order, and we also assert the following relations between the time points,

6. $t_1 < t_2$
7. $t_2 < t_3$
8. $t_3 < t_4$
9. $t_4 < t_5$
10. $t_5 < t_6$
11. $t_6 < t_7$
12. $t_7 < t_s$
13. $t_s < t_9$

We would like to be able to deduce that at time $t_9 > t_7$ Nikhil is at cricket practice, represented by *HoldsAt*(loc(cricketPractice11, nikhil53), t_9). To be able to do this we will need to add knowledge in the form of rules relating the events to the states. Such knowledge is called a set of *domain axioms*. For the above problem we need the following axioms. All variables are implicitly universally quantified. Axiom A_1 states that event *fillAir*(P, B) initiates the fluent *inflated*(B) at some time T . That is, fluent *inflated*(B) becomes *true* after time T as a consequence of action *fillAir*($P B$) which happens at time T .

A_1 : Initiates(fillAir(P, B), inflated(B), T)

Axiom A_2 states that if the preconditions of riding a bike, being at the source and the bike (tyres) being inflated, are *true* then the intention of riding the bike will succeed²⁰.

$A_2: ((\text{HoldsAt}(\text{inflated}(B), T) \wedge \text{HoldsAt}(\text{loc}(P, S), T)) \supset \text{Initiates}(\text{planToRide}(P, B, S, D), \text{worksPlanRide}(P, B, S, D), T))$

Axiom A_3 is known as a *trigger* axiom. A trigger axiom describes a set of fluents, in this example only one, that are triggers for events. When the trigger fluents become *true* the events are triggered automatically. Axiom A_3 states that if the fluent *worksPlanRide* holds then the *ride* event happens.

$A_3: \text{HoldsAt}(\text{worksPlanRide}(P, B, S, D), T) \supset \text{Happens}(\text{ride}(P, B, S, D), T)$

The reason we have set up these elaborate inferences is because in our story, we have only said that Nikhil has an intention or plan to ride the bicycle to the restaurant. His plan will fructify only if the necessary conditions hold, for example that his bicycle tyres remain inflated. The domain axiom A_2 is saying that if such conditions are satisfied then a plan to ride the bike is workable. A_3 says that if a plan is workable then it will happen.

Axiom A_4 states that if the *ride* event happens then the person P will be at the destination D . We have assumed this to happen instantaneously, but the reader is encouraged to write a temporal version of these axioms in which actions are durative.

$A_4: \text{Initiates}(\text{ride}(P, B, S, D), \text{loc}(P, D), T)$

Likewise if the *ride* event happens the person ceases to be at the source.

$A_5: (S \neq D) \supset \text{Terminates}(\text{ride}(P, B, S, D), \text{loc}(P, S), T)$

Axiom A_6 considers the preconditions for the eating at the restaurant plan to work, which we have taken only to be at the restaurant. In practice we will need more conditions like the customer has money.

$A_6: \text{HoldsAt}(\text{loc}(P, R), T) \supset \text{Initiates}(\text{planToEat}(P, R), \text{worksPlanEat}(P, R), T)$

Axiom A_7 is the trigger axiom for the eating event. Note that every intention (plan or action) that has preconditions will need such a trigger action.

$A_7: \text{HoldsAt}(\text{WorksPlanEat}(P, R), T) \supset \text{Happens}(\text{eat}(P, R), T)$

We assume that $\neg \text{Released}(f, t_0)$ holds for all fluents to start with. This means that fluents can change value only if they are affected by some events that happen.

The derivation of the $\text{HoldsAt}(\text{loc}(\text{Nikhil53}, \text{cricketPractice11}), t_9)$ for the narrative N_1 is as follows,

- | | | |
|-----|--|---------------------------------------|
| 14. | $\text{HoldsAt}(\text{inflated}(\text{bicycle39}), t_1)$ | From EC_1, A_1 and 2 |
| 15. | $\text{HoldsAt}(\text{inflated}(\text{bicycle39}), t_3)$ | From 14 and law of inertia |
| 16. | $\text{HoldsAt}(\text{loc}(\text{nikhil53}, \text{home86}), t_3)$ | From 1 and law of inertia |
| 17. | $\text{HoldsAt}(\text{worksPlanRide}(\text{nikhil53}, \text{bicycle39}, \text{home86}, \text{restaurant66}), t_3)$ | From 3, 15, 16, EC_1 and A_2^{21} |
| 18. | $\text{Happens}(\text{ride}(\text{nikhil53}, \text{bicycle39}, \text{home86}, \text{restaurant66}), t_3)$ | From 17 and A_3 |
| 19. | $\text{HoldsAt}(\text{loc}(\text{nikhil53}, \text{restaurant66}), t_3)$ | From 18 and A_4 |
| 20. | $\neg \text{HoldsAt}(\text{loc}(\text{nikhil53}, \text{home86}), t_3)$ | From 18 and A_5 |

21.	HoldsAt(loc(nikhil53, restaurant66), t_7)	From 19 and law of inertia
22.	HoldsAt(inflated(bicycle39), t_7)	From 15 and law of inertia
23.	HoldsAt(worksPlanRide(nikhil53, bicycle39, restaurant66, cricketPractice11), t_7)	From 5, 21, 22, EC_1 and A_2
24.	Happens(ride(nikhil53, bicycle39, restaurant66, cricketPractice11), t_7)	From 23 and A_3
25.	HoldsAt(loc(nikhil53, cricketPractice11), t_7)	From 24 and A_4
26.	HoldsAt(loc(nikhil53, cricketPractice11), t_9)	From 25 and Inertia

The above reasoning process shows that Nikhil is at cricket practice at time t_9 . But is that (fact) logically entailed from the given information? Remember that entailment means that the conclusion is *true whenever* the premises are *true*. It so happens that this is not so for the above statements. Because there might be other statements that are also *true* but which are not mentioned in the narrative. For example it is possible that an event occurred in which Lavanya deflates Nikhil's bicycle at time t_4 . Or the event that Nikhil went off to a movie at time t_8 , which is not mentioned in the narrative.

The astute reader would have noticed that we have glossed over the law of inertia and used it somewhat casually in the above "proof". In fact, if we want the value of a fluent to be carried forward from one time to another we need to use the Inertia Axiom (reproduced here again for convenience).

$$IA: (HoldsAt(f, t_1) \wedge (t_1 < t_2) \wedge PersistsBetween(t_1, f, t_2) \wedge \neg Clipped(t_1, f, t_2)) \supset HoldsAt(f, t_2)$$

To move from line 14 to line 15 we would need to apply the following instance of the above axiom,

$$\begin{aligned} & HoldsAt(inflated(bicycle39), t_1) \wedge (t_1 < t_3) \wedge \\ & PersistsBetween(t_1, inflated(bicycle39), t_3) \wedge \\ & \neg Clipped(t_1, inflated(bicycle39), t_3) \\ & \supset HoldsAt(inflated(bicycle39), t_3) \end{aligned}$$

A transitivity rule can be added to take care of the formula $(t_1 < t_3)$. We need to show that $PersistsBetween(t_1, inflated(bicycle39), t_3)$ and $\neg Clipped(t_1, inflated(bicycle39), t_3)$ hold before we can show that the bike is inflated at time t_3 . The former says that the $ReleasedAt(inflated(bicycle39), t)$ did not happen for any time t between t_1 and t_3 (the fluent was not released from the commonsense law of inertia), while the latter says that no event happened during that time which would have made the fluent *false*.

Likewise for the fluent $loc(Nikhil53, cricketPractice11)$ to be *true* at time t_9 , or the formula $HoldsAt(loc(Nikhil53, cricketPractice11), t_9)$ to be entailed, one has to make an *assumption* that nothing else happens that interferes with the chain of events that make the fluent *true*, and nothing happens after it has been made *true* that makes it *false*.

Such assumptions are characteristic of non-monotonic reasoning where one deals with statements that can change their truth value. We do not need to make

such circumscribing assumptions in classical (mathematical) logic because once a conclusion is deduced the addition of other statements does not change its truth value. Hence mathematical reasoning can happen in an open world, but deductive reasoning about change can effectively happen only under assumptions that it is a closed world. That is, we assume that we know everything that is relevant, and if we don't know something to be *true*, it must be either *false* (negation by failure) or irrelevant to the conclusion we are interested in.

Box 13.1 : The Frame Problem

Consider a robot carrying out some task autonomously. Assume that it maintains its set of beliefs about its world as a set of propositions. Given that it is planning some actions to be done, how does it update its representation to account for the changes that happen as a result of its actions?

Of course some changes are intended and are captured in its representation of actions. For example, double clicking on file icon in one's computer will "open" the file. Or turning the faucet will make the water flow from the tap. Or shooting with a gun will result in the death of the target.

The question is *what does not change* as a consequence of a given action? Is there a succinct way of asserting that? (Hayes, 1987). For example, if one fired a gun a bird sitting on a nearby tree might take flight. Or turning the faucet on will (somehow) turn on a light as well.

John McCarthy and Patrick Hayes (1969) called this the *Frame Problem*. To use the analogy of drawing cartoon animations by hand, what does not change from one frame to the next, and can be carried forward? They illustrate this with an example in which an agent looks up a telephone directory intending to call a friend over the phone. How is one sure that the action of looking up the directory has not (somehow) made the phone vanish?

One can see that the solution to this problem of logical reasoning is to somehow assert that actions have no unstated effects, and also that no unstated actions have happened.

Steven Hanks and Drew McDermott (1987) pose the *Yale shooting problem* in which a gun is loaded and fired after a time interval. How can one conclude that the target is dead as a consequence?

From a philosopher's point of view, the question is how can an agent ever be sure that it has updated its beliefs to reflect all the changes that are a consequence of its actions? Drew McDermott's response to this is that even humans are unable to guarantee this and can make mistakes (McDermott, 1987).

We have to assume that everything that matters has been stated explicitly. *Given* such an assumption and given the facts, the conclusion arrived at by the derivation is then indeed entailed by what is stated (including the assumption). Thus under such an assumption both *PersistsBetween*(t_1 , inflated(bicycle39), t_3) and \neg *Clipped*(t_1 inflated(bicycle39), t_2) will hold, and we will be able to conclude that bike is inflated at time t_3 at line 15.

For the conclusion to be an entailment it means that it must be *true* in all models of the set of *EC* formulas describing the domain, the *EC* axioms, and situation and the events. A model is a combination of a domain and mapping of *EC* statements and terms to relations and elements in the domain. The following additional assertions need to be made.

Unique Name Axioms

The first thing that one must assert is that terms or fluents that are named differently actually map to different elements in the domain. For example the fact that “fillAir” and “eat” refer to distinct events. Such assertions are typically made as unique name axioms (UNA). Thus we should say that,

UNA(loc, inflated, worksPlanRide, worksPlanEat)
UNA(fillAir, planToRide, planToEat, ride, eat)
UNA(nikhil53, home86, bicycle39, restaurant66, cricketPractice11)

The meaning of a UNA statement is that all its arguments are distinct.

Predicate Completion

The next “fact” we must assert is that the events have no unstated effects, and that no unstated events have actually happened.

Completion of the *Initiates* predicate is done by listing all known events that initiate fluents and the corresponding fluents. Thus if *Initiates*(*e*, *f* *t*) is *true* then *e* and *f* can *only* take one of the following pairs of values. Observe that the *e*, *f* and *t* are universally quantified “variables” of the EC, and their arguments variables from the domain.

$$\begin{aligned} \text{Initiates}(e, f, t) \Leftrightarrow & (e = \text{fillAir}(P, B) \wedge f = \text{inflated}(B)) \\ & \vee (e = \text{planToRide}(P, B, S, D) \wedge f = \text{worksPlanRide}(P, B, S, D)) \\ & \vee (e = \text{ride}(P, B, S, D) \wedge f = \text{loc}(P, D)) \\ & \vee (e = \text{planToEat}(P, R) \wedge f = \text{worksPlanEat}(P, R)) \end{aligned}$$

The above formula states that if an event *e* is initiating a fluent *f* then it must be the *fillAir* event that results in the *inflated* fluent becoming *true*, or the *planToRide* event that results in *worksPlanRide* fluent, the *ride* event that results in the *loc* fluent, or the *planToEat* event that results in the *worksPlanEat* fluent with appropriate values. No other effect of any action can happen.

Likewise the only case we know when an event terminates a fluent is

$$\text{Terminates}(e, f, t) \Leftrightarrow (e = \text{ride}(P, B, S, D) \wedge f = \text{loc}(P, S), T)$$

When we talk of the *Happens* predicate, we have to be a bit more specific about the actual instances. We want to eliminate event instances that no one has said have happened, without eliminating the event (type) itself. The only (instances) of events that have been stated *explicitly* to happen are,

$$\begin{aligned} \text{Happens}(e, t) \Leftrightarrow & e = \text{fillAir}(\text{nikhil53}, \text{bicycle39}) \wedge t = t_1 \\ & \vee e = \text{planToRide}(\text{nikhil53}, \text{bicycle39}, \text{home86}, \text{restaurant66}) \wedge t = t_3 \\ & \vee e = \text{planToEat}(\text{nikhil53}, \text{restaurant66}) \wedge t = t_5 \\ & \vee e = \text{planToRide}(\text{nikhil53}, \text{bicycle39}, \text{restaurant66}, \text{cricketPractice11}) \wedge t = t_7 \end{aligned}$$

However, the above restriction will not fit the bill because along with events that no one has stated to happen it also throws out events that happen due to their trigger conditions becoming *true*. One needs to allow those actions that are logically consistent with the set of formulas that we have.

This is a little bit trickier. What we really want to say is that only those events happened that are either explicitly stated or that follow logically from what is stated. This is precisely what Circumscription does.

Circumscription

Circumscription is a method of default reasoning devised by John McCarthy (McCarthy, 1980; 1986), (Lifschitz, 1985; 1994). It is an approach to default reasoning that is aimed at making plausible, or defeasible, inferences. The basic idea behind Circumscription is to find the set of formulas that hold in all minimal models of a given set of formulas. In doing so one may choose the predicates that one wants to minimize.

If Γ is a formula containing the predicate symbol ρ , then the circumscription of ρ in Γ written as $\text{Circ}[\Gamma \rho]$ is the formula in second order logic,

$$\Gamma \wedge \neg(\exists \varphi(\Gamma(\varphi) \wedge \varphi < \rho))$$

which should be read as: The statement Γ is *true*, and there exists no predicate φ of the same arity as ρ such that $T(\varphi)$, the formula Γ with every occurrence of ρ is replaced by φ , is *true*, and φ implies ρ but is not equivalent to ρ . In other words in every model of the circumscription the set of instances of ρ are consistent with (the given) Γ , and no subset of the instances is consistent with Γ . Equivalently the circumscription admits only those instances of ρ that are necessarily entailed by the given set of facts Γ . We shall look at Circumscription again when we look at default reasoning in Chapter 17.

Let Σ be the conjunction of the domain axioms including the *Initiates* and the *Terminates* formulas, and let Δ be the set of *Happens* statements including the trigger events, let EC be the axioms of event calculus, and U be the set of unique named assumptions, then we can state that

$$\begin{aligned} &\text{Circ}[\Sigma; \text{Initiates}, \text{Terminates}, \text{Releases}] \wedge \text{Circ}[\Delta; \text{Happens}] \wedge EC \wedge U \\ &\models \text{HoldsAt}(\text{loc}(\text{nikhil53}, \text{cricketPractice11}), t_9) \end{aligned}$$

The detailed proofs are beyond the scope of this text. The interested reader is referred to (Lifschitz, 1985; 1994), (Mueller, 2006) and (Shanahan, 1995; 1997). Our focus has been on representation in this chapter, and we are looking at how one can represent the relationship between events and fluents in a logic based setting.

13.5.4 A Simpler Narrative N₂

Why did we use the action *planToRide*(P, B, S, D) instead of the action *ride*(P, B, S, D) directly? Suppose we had given the narrative stating the events that had actually happened as follows,

- I. *Happens*(*fillAir*(*nikhil53*, *bicycle39*), t_1)
- II. *Happens*(*ride*(*nikhil53*, *bicycle39*, *home86*, *restaurant66*), t_3)
- III. *Happens*(*eat*(*nikhil53*, *restaurant66*), t_5)
- IV. *Happens*(*ride*(*nikhil53*, *bicycle39*, *restaurant66*, *cricketPractice11*), t_7)

then the conclusion requires a smaller set of assumptions, only for the period between t_7 and t_9 . This is because the events that have been stated as happened *must* have happened (because we assume our knowledge base is consistent). Thus we can be sure that the last *ride* event initiates the fluent *loc*(*Nikhil53*, *cricketPractice11*) at time t_7 , and we only have to assume that nothing happens after that.

The earlier narrative N_1 stated that Nikhil *intends* to ride his bike to the restaurant, *intends* to eat, and *intends* to go to cricket practice. The *intention* of the actor is known. Using Circumscription, we can assume that these plans will indeed work, since we have no knowledge of any other events that could have possibly interfered with intention, and we can “conclude” that he is at cricket practice at time t_7 . This conclusion is a default or tentative or defeasible conclusion, valid only under the assumption that all relevant events and their effects have been explicitly stated. However, the story N_1 does allow for a new event to be added to the knowledge base, for example that Lavanya deflates his bike at time t_4 .

V. Happens(deflate(lavanya7, bicycle39), t_4)

Our knowledge base will still be consistent. The conclusion, however, that Nikhil is at will no longer be entailed. If we had used the second narrative N_2 , then the addition of the new *fact* of Lavanya deflating the bike would have made the knowledge base inconsistent. This is because an action with preconditions in fact implies the preconditions when it happens (see Chapter 10). Thus, the following is an axiom,

$$A8: \text{Happens}(\text{ride}(P, B, S, D), T) \supset \text{HoldsAt}(\text{inflated}(B), T)$$

which would have led us to conclude both that the bike is deflated (thanks to Lavanya) as well inflated (since the bike ride happened) at the same time point t_7 , which is a contradiction²². In the classic Yale shooting problem that has been discussed in the event calculus literature, the *intention* of killing is represented by the *shoot* action, and the *implicit* killing event uses the *loaded* fluent as a precondition. Thus, the intention of killing (shooting) leads to the actual killing event only when the gun is loaded. More importantly, the *shoot* action is possible even without the precondition being *true*, and can be executed even when the gun is *not loaded*, though it does not result in the *implicit event of killing*. In our description, we have separated the intention as the *planToRide* action, which leads to success only when the bike is inflated and the person is at the source, but when it does work it leads to the *ride* event happening via a trigger axiom. If the preconditions do not hold the *planToRide* event can still *happen*, though it would not *result in* the *ride* event happening.

Random Events

Deflation of the (tyres of the) bike need not only happen due to adversarial action. It could happen due to the tyres coming into contact with a sharp object such as a nail or a pointed stone, which may or may not result in deflation. This could be modeled using the *Releases* predicate as follows

$$\text{Releases}(\text{hitNail}(B), \text{inflated}(B), T)$$

Now in some models the bike would be inflated and in some it would not. But since *ReleasedAt*(*inflated*(*bicycle39*), *t*) would happen at whatever time *t* the bike hits the nail it will no longer be possible to *deduce* that the bike (tyre) is inflated at a later point of time, and also therefore none of the conclusions that depend upon the fluent being *true*.

This could also be modeled using a random variable *Punctured*, called a determining fluent, that could take any random value, *true* or *false*, as follows

$$(HoldsAt(inflated(B), T) \wedge HoldsAt(Punctured, T)) \supset Terminates(hitNail(B), inflated(B), T)$$

Being a random variable *Punctured* is not governed by the commonsense law of inertia. Then in some models *Punctured* will be *true* and in some it will be *false*, and one cannot come to a definitive conclusion about whether the bike is inflated or not.

Conceptually the first approach says that hitting a nail releases the fluent *inflated* from the common-sense law of inertia, and thus one cannot infer whether or not it holds after that. The second approach says that there is a variable, which if it happens to be *true*, will result in the bike tyre getting deflated.

Trajectory

The *Trajectory* predicate is used to relate two related fluents one of which is *Initiated* by some event in a changing world. In the above example assume that along with *loc(Person, Place)* we have a fluent *dist(Person, Place, Distance)* in a one dimensional world, where *Distance* is of the real number sort. Assume that the bicycle riding event has a duration, and during that duration fluent *ridingBike(Person)* is *true*. Then we can express the distance the person has travelled at any given time by a function of the speed. Let us say that Nikhil rides his bike at *S* metres/second. Then we can write the following expression,

$$\begin{aligned} & (HoldsAt(loc(nikhil53, home86), t_3) \\ & \wedge Happens(ride(nikhil53, bicycle39, home86, restaurant66), t_3) \\ & \wedge (t_3 < T) \\ & \wedge \neg Clipped(t_3, ridingBike(nikhil53), T)) \\ & \supset HoldsAt(dist(nikhil53, home86, S * T), t_3 + T) \end{aligned}$$

The above expression says that if Nikhil was at location Home at time t_3 , and started riding his bike (at a constant speed of *S* meters/sec) then at time $t_3 + T$ he will be $S * T$ meters away from home, provided the *ridingBike* fluent was not terminated in the intervening period.

The trajectory axiom allows us to say that in a succinct form as follows:

$$\begin{aligned} & HoldsAt(loc(nikhil53, home86), t_3) \\ & \supset Trajectory(ridingBike(nikhil53), t_3, dist(nikhil53, home86, S * T), T) \end{aligned}$$

which may be generalized as,

$$HoldsAt(loc(P, L), t_0) \supset Trajectory(ridingBike(P), t_0, dist(P, L, S * T), T)$$

which says that if a person *P* is at a location *L*, and some event happens that initiates the *ridingBike(P)* fluent at time t_0 , then the distance of the person from *L* after time *T* is given by $S * T$.

This brief introduction to the event calculus has given us a glimpse of an approach for logical reasoning in a changing world, and the issues involved. Like *FOL*, the event calculus gives us a *mechanism* for making deductions about what becomes or remains *true* as a result of events happening. The changing properties are represented as fluents that are initiated or terminated by events.

13.5.5 Knowledge and Belief

How do we model the knowledge held by agents? If one is to model an agent having knowledge of some relations (“Sydney knows that Charles loves Lucie”) or some events (“Akira knows that the flight had taken off”) or even beliefs (“Drona believed that Aswathama had been killed”), then one has to add the corresponding formulas as reified objects that can be the argument to the formulas for knowledge and belief.

One of the first approaches, employed by Hintikka (1962), was to introduce a modal operator $K_a(a)$ to stand for “Agent a knows a ”. The semantics of the operator is the commonly accepted Kripke’s “possible worlds” semantics, that a is *true* in all possible worlds compatible with a ’s knowledge. Adopting a logical formalism one can reason about who knows what and (assuming that the agents are equally adept) come up with answers to puzzles involving common knowledge (see for example (Halpern and Moses, 1984), (Halpern, 1984), (Lehmann, 1984), (Stewart, 1998), and (Moses et al., 1986)).

We will not foray into *Modal Logics* here, but rely on the observation by McCarthy (McCarthy, 1986) that the many world semantics can be achieved in practice by imagining many worlds, parameterized by situation names, and then relying on the truth functional semantics of *FOL*. Moreover, we move away from the deductive form of reasoning and look at other ways in which knowledge can be represented and usefully employed. We begin by looking at knowledge and belief as modeled by the *Conceptual Dependency (CD)* theory of Roger Schank.

The question of *what* fluents an agent needs to define is still an open question. The work in logic focuses more on the reasoning aspect and assumes that the predicates and actions have already been defined. For most illustrative purposes researchers tend to use predicates and action names chosen in an *ad hoc* manner from natural languages. This brings in the associated richness of natural language vocabularies, but along with that also the problems of dealing with synonyms, hyponyms, hypernyms, homonyms, meronyms, paronyms and so on. The use of natural language words also leaves the problem of translation between different languages open.

One can ask is whether the knowledge represented by an agent is dependent upon the language used by the agent or is it represented independent of it in some “conceptual” manner. A few thousand years ago when two different humans in different parts of the world spied an apple on a tree and reached out for it, was there anything common in their representation and processing? Did William Shakespeare and James Joyce have access to a larger set of concepts or a more powerful faculty of describing them?

There are two reasons to move away from a linguistics based representation. The first, the fact that there are many different languages talking about the same things suggests that one should be able to look for a common representation. The second is that even with a chosen language there is a surfeit of richness of expression allowing different ways of saying the same thing. This may contribute to the aesthetic demands of literature but compounds the difficulty in automatic reasoning. If we could come up with a core set of conceptual representations then the task of reasoning would be simplified because the vocabulary would become smaller. Such an interesting exercise was done through the seventies and early eighties on a large scale.

13.6 Conceptual Dependency Theory

A novel approach to knowledge representation was the work emerging out of Yale University in the seventies. Spearheaded by Roger Schank many interesting natural language understanding programs were built around his Conceptual Dependency (CD) theory of representation. The work was done in the context of story understanding and began with representing everyday actions using a core set of conceptual *actions* and *states*. The CD theory represents all states and events using a small set of primitive concepts, and maps incoming natural language sentences into structures of this canonical representation. The goal was that understating statements in *any* natural language would result in the *same* representation, and as a corollary, generators for different languages could produce output in that language. The reasoning and inferences needed for understanding can then be done at canonical level, and can circumvent having to deal with a vast number of linguistic terms.

We look at the basic idea behind CD theory here, and in the next chapter we will look at some knowledge structures that were built using the CD representation. The structures focus on the contexts in which events are happening, and also the relations between goals, plans and actions of an actor.

The *Conceptual Dependency theory* describes the world in terms of four kinds of syntactic constituents.

PP : *Picture producers* These correspond to noun phrases and stand for elements of the domain, including Actors and all inanimate objects.

PA : *Picture aiders* These correspond broadly to adjectives, and are used to describe properties of PPs.

ACT : *Actions* These are conceptual actions. Very often linguistic verbs stand for conceptual actions but, as we shall see, this is not always so. Further the CD theory assumes a small set of primitive ACTs that are used to compose complex actions. ACTs are carried out by actors.

AA : *Actions aiders* These describe the properties of the ACTs and roughly correspond to adverbs. They may also be used to describe properties of events.

The world is described in terms of *events* and *states* and the relations between them. Events can be either the ACTS of an actor or a change of state. The latter are called state change events. The relations between the different constituents are captured as *dependencies*. Some dependencies are two way dependencies, and they form *conceptualizations*, which correspond to sentences in logic, and can stand by themselves.

The following dependencies can occur. In keeping with the notation used by the Yale group we depict conceptual structures graphically using arrows to capture dependencies. While implementing them one will convert them into formulas of FOL.

1. **Actors can act.** For example, the event “Harry ate the pear” could be represented by (INGEST ACTOR (HARRY) OBJECT (PEAR REF (DEF))). At an abstract level there is a two way dependency between an Actors and an ACT, which we can represent as $PP \Leftrightarrow ACT$

$PP \longleftrightarrow ACT$

2. ***Picture producers can be described by picture aiders.*** Picture aiders are connected to PPs by three line arrows. This could be used for example to express the statement “Jaidev is tall”. $PP \rightleftarrows PA$

PP \longleftrightarrow PA

3. **Objective case.** ACTs can have objects. For example, “Saveri lent *the book* to Aditi”.

ACT $\xleftarrow{\circ}$ PP

4. **Directive case.** ACTs can be associated with a sense of direction. For example “Nikhil went *towards the restaurant*”

ACT \xleftarrow{D} $\begin{cases} \rightarrow \text{LOC} \\ \leftarrow \text{LOC} \end{cases}$

5. **Recipient case.** ACTs can have recipients. For example in sentence “Saveri lent the book to Aditi”, Saveri is the lender and Aditi the borrower.

ACT \xleftarrow{R} $\begin{cases} \rightarrow \text{PP} \\ \leftarrow \text{PP} \end{cases}$

6. **The objective case can take a conceptualization as the object.** For example this might occur when the ACT is a communication act and one is telling a story. For example, “Mahathi told her father that *Praveen had plucked the flower*”. Observe that we have now moved higher from strict FOL and one of the arguments to the relation is a conceptualization itself. This is similar to what one would represent such statements with logics of knowledge and belief.

ACT $\xleftarrow{\circ}$ \updownarrow

7. **The instrumental case.** ACTS may have instrumental ACTS. For example, “He sent her the message by *writing it on a piece of paper and throwing it to her*”.

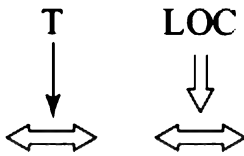
ACT \xleftarrow{I} \updownarrow

8. **PPs can mark conceptualizations, and in turn may be described by them.** For example, “*The ground* where the fight between the boys took place”, or “*The girl* in the field full of daffodils”.

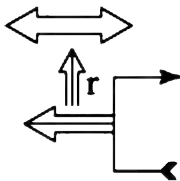
PP \updownarrow
 \longleftrightarrow PP \updownarrow
 \longleftrightarrow

9. **Conceptualizations can have a time marker or a location marker.** In the

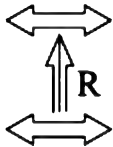
sentence ““Harry *ate* the pear in *the park*” the past tense would be indicated by the marker *P* and “*the park*” would mark the event. A set of symbols could represent different conceptual tenses, for example future (*f*), transition (*t*), continuing (*k*), interrogative (?), negative (*/*), potential (*c*), present (no marker).



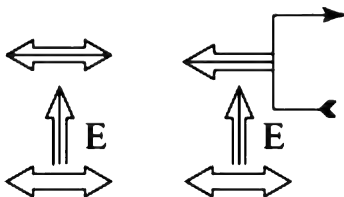
10. **Conceptualizations can result in state change events.** For example, “They converted the grapes into juice by dancing upon them”. Note that the arrow points from the effect to the cause, to signify that the state change is *dependent* upon the action event.



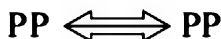
11. **Conceptualizations can serve as a reason for other conceptualizations.** For example, “Janet opened the door for David and he walked into the movie hall”, or “The mice bit the ropes enabling the fox to escape from the trap”.



12. **A state or state change may enable a conceptualization.** Very often people do things because of a given state or a changing state. For example, “He was angry and kicked his foot against the wheel”, or “Since it started to rain he ran out to pick up the laundry”.



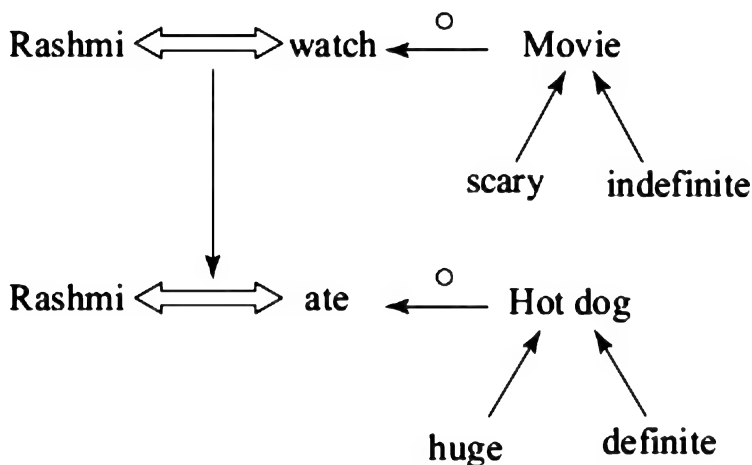
13. **A PP can describe another PP.** For example “Barack Obama is the President” or “The doctor was the thief”.



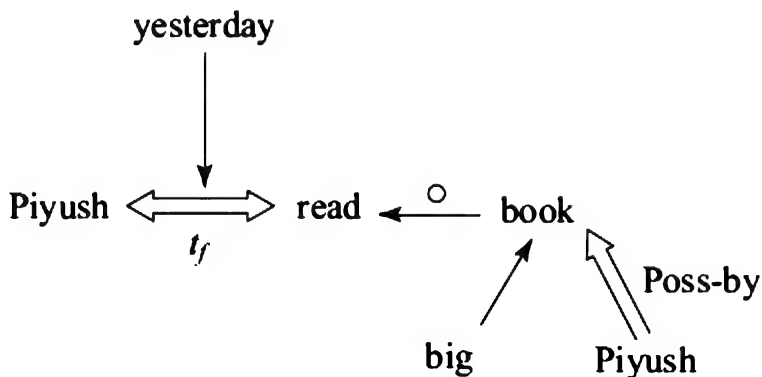
14. **ACTS can be further described by Action aiders.** “He was walking very fast”.

ACT
↑
AA

15. **Attributive dependency.** The single arrow is used when something is an attribute of something else. The following illustrates this with an example. "Rashmi ate the huge hot dog while watching a scary movie". In the diagram the movie has two attributes, one that it is indefinite (because it is "a movie") and the other that it is scary. Likewise the hot dog has two attributes one that it is "the hot dog" and the other is that it is huge. Finally, the event of watching the movie is an attribute denoting when the event of eating the hot dog occurred.



16. **Prepositional dependency.** A double arrow is used to denote dependencies like "possessed by", "contained in", "containing". For example, "Piyush finished reading his big book yesterday" would be represented as, yesterday



In the above figure, the label t_f signifies that the act of reading has terminated, while the attribute yesterday marks the entire event. The fact that the book belongs to Piyush is shown as an attributive dependency.

13.6.1 Conceptual Actions and States

In the illustrations above we have freely used action and state names borrowed from the English language. However, one of the objectives of the *CD* theory was to get to the meaning or semantics behind natural language utterances. Understanding a natural language utterance here means arriving at a possibly language independent semantic representation from the utterance. This representation should be canonical in the sense that the many different ways of “saying the same thing” that the richness of language allows us²³ should all map onto it. This does not debar one from using words from a language as names of concepts, as long as one is conscious of the overloading.

The *CD* theory, which has been used to describe every day actions, uses a small set of state variables, and a small set of conceptual acts.

The state variables take numerical values, and similar language words or phrases map to different values of the variables. Some of the variables are described below. We have taken some illustrative mappings of words from the English language. One can extend the system to define mapping words from other languages as well. It would then serve as a common representation of meaning for different languages.

health The value of this variable varies from -10 to 10. The different English words that could be mapped to it are, with their values, dead (-10), gravely ill (-9), sick (-9 to -1), under the weather (-2), all right (0), tip top (7), and perfect health (10). A person falling ill could be modeled as going from a state of say *health* = 4 to say *health* = -3.

anger take values -10 to 0. Some examples are, furious (-9), enraged (-8), angry (-5), irked (-3), upset (-2) and calm (0).

joy goes from -10 to +10. Examples, catatonic (-9), depressed (-5), upset (-3), sad (-2), OK (0), pleased (2), happy (5), ecstatic (10).

fear goes from -10 to 0. For example, terrified (-9), scared (-5), anxious (-2), calm (0).

hunger goes from -10 to +10. Examples: starving (-8), ravenous (-6), “could eat a horse” (-5), hungry (-3), no appetite (0), satisfied (3), full (5), stuffed (8), and satiated (3 to 10)

disgust goes from -10 to 0. Example words that are mapped on to it, nauseated (-8), revolted (-7), disgusted (-6) and bothered (-2).

surprise goes from 0 to 10. Examples: surprised (5), amazed (7), astounded (9).

consciousness goes from 0 to +10. Examples: unconscious (0), asleep (5), awake (10), “higher drug consciousness” (> 10).

Some states may be expressed as conjunctions of the “primitive” states, for example shocked = surprise (6) \wedge disgust (-5).

These are states of human beings. Likewise other entities could have states too. One could also account for modeling cultural differences in the use of language by a community specific mapping onto the values of a variable. For example, if a community is known to understate things, then the statement “I’m a bit upset with your behaviour” or “I’m a little unwell” could be mapped appropriately

to the values to compensate for the linguistic understatement. On the other hand, if a community is known to overstate matters then their statements could be mapped onto the values with a pinch of salt.

Human beings are creatures of the mind. The mental state of a human being usually has a significant influence on behavior, and has been a subject of intense study in psychology and related fields. A plethora of self help books teach us how to control our mind and emotions, and techniques like meditation and deep breathing are often recommended to people who tend to get overly excited. Is there a “complete” set of variables that can be used to describe all emotional states of humans? Is there a set of primitives in terms of which one can describe all emotional states? It should be an interesting quest for those aiming to model emotions.

13.6.2 The Rasa Theory

One of the earliest attempts in characterizing human emotion was in Bharata’s ancient classical work called the *Nāṭyaśāstra*, (the science of drama, dance and music) about two millennia ago (Mishra, 1964), (Vatsayayan, 1996). Written in Sanskrit, the text consists of 6, 000 *sutras*, or verse stanzas, incorporated in 36 or 37 chapters²⁴. The theory of emotions is called the *Rasa* theory (*rasa* means juice, so in a way it is like extracting the essence of human emotions). The motivation of the work, as the name suggests, was to identify “*the dominant emotions and permanent emotions in the heart of every human being*” (Sharma, 2003). The *Nāṭyaśāstra* identifies eight *sthayibhavas* or basic emotions:

- *Rati* (Love, amorous and romantic)
- *Hasya* (Mirth, the capacity to enjoy a comic situation)
- *Soka* (Sorrow, or grief, that arises due to a loss)
- *Krodha* (Anger, caused by injustice to oneself or others)
- *Utsaha* (Energy, the enthusiasm to do something)
- *Bhaya* (Terror, fear of the dangerous)
- *Jugupsa* (Disgust, repulsion)
- *Vismaya* (Astonishment, on encountering something unusual)

These eight basic emotions (*sthayibhavas*) are often identified themselves as the eight *rasas*. These stationery emotions often give rise to auxiliary fleeting emotions, *sanchari bhavas*, which reinforce and support the basic emotions. The expression of these *sanchari bhavas* by a skilled performer are experienced and absorbed by the receiver (*rasika*) as *rasa*, the essence of feeling. The *rasas* that can be experienced have been described as follows.

Shringar Rasa Arising out of *Rati*, it is one of the oldest emotions known since a human being espied an attractive mate. Bharata classifies the feelings into two kinds, *sanyoga shringar*, love in union, and *viyoga shringar*, love in separation. Shringar *rasa* probably forms the core subject of a majority of Indian dance dramas.

Hasya Rasa Is evoked by the jester’s funny behaviour in a drama. It was also the *rasa* experienced by Draupadi in the Indian epic *Mahabharata* when she saw Duryodhana fall unto a pool of water, leading eventually to the great war.

Raudra Rasa The feeling of anger arising out of *krodha*, was probably the emotion felt by Duryodhana when he saw Draupadi break into peals of laughter. It is a common emotion invoked in modern cinema, especially Indian cinema, when the protagonist yields to his²⁵ rage and overcomes injustice against all odds.

Veer Rasa Or the heroic sentiment (in the hearts of the audience) arises chiefly out of *utsah*, or enthusiasm. This could be due to good unselfish deeds being depicted, or evoked by valour against enemies.

Bhayanak Rasa The sentiment of fear, when confronted with dreadful and terrible creatures like monsters, witches and ferocious animals.

Karuna Rasa Arises out of grief or *soka*, is the foundation of a tragedy. It may arise out of loss of dear ones, an epidemic, an earthquake, or a tsunami.

Veebhatsa Rasa Is the dominant emotion of disgust, making the *rasika* turn her face away. Often invoked by a gruesome scene, but could also be a result of someone's abominable behaviour.

Adbhut Rasa The sense of wonder one feels, for example, while watching Carl Sagan talk evokingly of the cosmos, or when a child beholds a blossoming valley of flowers, so well depicted in many an animation film.

Shanta Rasa Peace, quietude, detachment, the feeling of calmness, was not included by Bharata amongst the eight rasas and discussed separately, perhaps because it is the opposite of drama. In the *CD* theory we might describe it as $\text{surprise}(0) \wedge \text{disgust}(0) \wedge \text{fear}(0) \wedge \text{anger}(0) \wedge \text{consciousness}(>0)$.

Vatsalya Rasa Or parental love, was probably added later to the existing rasas. The objects of parental love are often children, who evoke these feelings in adults by their carefree laughter and innocent and disarming manner.

The nine *rasas* are known as the *navarasas* and probably form a significant part of the curriculum of any Bharatanatyam student.

13.6.3 Conceptual Dependency ACTs

Rather than use the plethora of verbs in natural languages to stand for actions, Roger Schank's *CD* theory used a small set of conceptual actions. These conceptual actions could be combined in different way into to build conceptualizations that could capture real world events. In fact the dictionary built for words from the natural language contains such conceptualizations. The exact number of ACTs in the *CD* theory varied between eleven and fourteen. We present one such set, and illustrate how they can be used for knowledge representation.

The *CD* ACTs are,

1. ATRANS, or Abstract Transfer The transfer of an abstract relationship such as possession, ownership or control. ATRANS would be instrumental in expressing concepts like give, take, buy, gift, receive, snatch, rob, and steal.

2. PTRANS, or Physical Transfer Transfer of physical location of an object. It would be used to model all movement words like go, visit, emigrate, climb, deliver furniture, send a letter, and walk.

3. PROPEL Or the *application* of a force to an object, regardless of whether the object is PTRANSed or not. Words that would need the PROPEL ACT for their representation are push, pull, throw, kick, hit, punch, caress, and hammer.

4. MOVE Or the movement of a body part of an animal by *that* animal. Often an instrumental act for the PTRANS act or the PROPEL act. Animate actors can move body parts resulting in some other conceptual act. For example, “He walked to the canteen” is conceptually a PTRANS using MOVE (foot) as an instrumental act.

5. GRASP The grasping of an object by an actor. This would be used by verbs like grab, let go, and throw. Observe that the use of a termination time marker would signify ungrasping.

6. INGEST To take in, including words like eat, drink, smoke, gobble, swallow, sip and breathe.

7. EXPEL Includes expulsion from the body. Spitting, urinating, gargling, sweating, exhaling, and even crying may need the EXPEL act.

8. MTRANS Or Mental Transfer. The transfer of information between animals or within an animal. Thus words like saying, telling, narrating, emailing, reciting, writing, reading, and SMSing would involve mental transfer. MTRANS is also used to create a folk psychology model of the mind or the brain. The (human) memory is modeled to have partitions – CP (conscious processor), IM (immediate memory), STM (short term memory) and LTM (long term memory). Thus, verbs like remembering, forgetting, understanding, seeing may involve movement or the lack of movement between different parts of the memory. One significant difference between PTRANS and MTRANS is that the “object” of transfer ceases to be at source in PTRANS, but not so in MTRANS.

9. MBUILD The construction or synthesis by an animal of new information possibly from old information. This could be used to model verbs like decide, conclude, imagine, consider, infer, and deduce.

10. SPEAK The actions of producing *sounds*. Humans often use it as an instrumental act for MTRANS. English words that would need SPEAK are say, play music, purr, scream, roar, growl and whisper.

11. ATTEND (Sense Organ) The action of attending or focusing a sense organ towards a stimulus. Also an instrument to MTRANS. To see is to MTRANS to CP from eye by instrument of ATTEND(eye) to object. Likewise to hear or listen is to ATTEND(ear).

We look at some examples to illustrate the use of *CD* acts and states to represent some conceptualizations expressed in English sentences.

13.6.4 Conceptualizations

Conceptualizations are representations in the language of conceptual dependency. While each *CD* act corresponds to some linguistic verb, the dependency relations allow us to combine different combinations to construct complex representations. We look at examples of English language sentences and their *CD* representations.

When we *believe* something then that conceptualization is resident in our immediate memory. For example, “I believe that Kasparov is brilliant” would be

represented as shown on top in Figure 13.10. This representation is a state description. In terms of logic this is like saying that the formula *Brilliant(Kasparov)* is an element in my set of beliefs.

We introduce the act CONC to correspond to the active notion of “thinking”, which could be a short form for something like MTRANSing something to or even within the CP (conscious processor). This is a more conscious or active way of thinking. Then a sentence like “I think that Kasparov is brilliant” could be represented as shown in the bottom part of the figure.

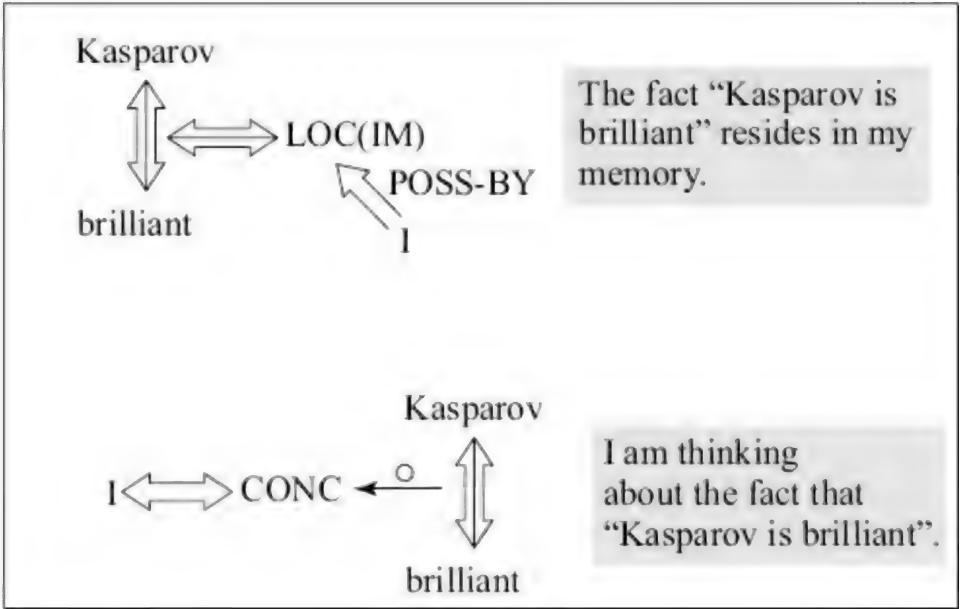


FIGURE 13.10 Two sense of the word “believe”

The reader might object that the two statements are in fact saying the same thing in a different way. That might be true, since the latter is often used to mean the former. But that does not belie the fact that the two *representations* are conceptually distinct. One is a description of a *state* of belief, the other represents a conscious thought process, an act. People often fail to find the best phrases to express themselves accurately, and could well do with the advice the March Hare gave to Alice (in Wonderland), “Then you should say what you mean”.

The reader must also realize that a compact symbolic account of thought processes as with the *CD* theory does not claim to explain the thought process in the cognitive science sense. In all likelihood, we *do not* move around formulas when we talk and think. But this is merely a way to create models that might help explain some aspects of the behaviour, and contribute to building useful computer programs. One could go into finer and finer level of detail to try and represent things more and more accurately. For example I might want to say that I came to realize that Kasparov is brilliant after I saw his game against Anand, and since then I believe that he is brilliant. This could possibly be represented as in Figure 13.11.

Observe that we are still taking liberties by using actions like “Play Chess”, and states like “brilliant” but that should not take away anything from the basic idea. We could have tried to model the process in greater detail by introducing movement to the CP and then to IM and then to LTM, and also introduce an instrumental act of

that “the butcher killed the lamb” we could have brought our world knowledge to fore (how can a program do that?) and imagined how the act would have been done. But if we look at the cockroach killing sentence itself, it does not specify how the act was done. Likewise consider the statement, “*Adora moved the television to the corner*”. This one sounds suspiciously like the act MOVE, though it is really a PTRANS. However, the statement does not say anything about a conceptual act. Again, perhaps she pushed the table on which the TV set was lying, or perhaps she asked the carpenter, or her sister, to move it, which means she actually did an MTRANS using the SPEAK act. In both the examples, we don’t really know what Adora did, but we do know that she did *something*, as a result of which the stated change occurred. We should therefore model it as shown in Figure 13.12.

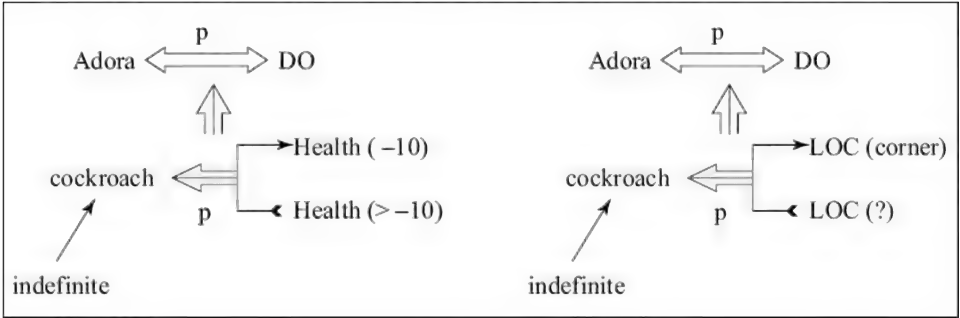


FIGURE 13.12 State change verbs from language are causal relations conceptually.

The DO action is a kind of a variable action which may be interpreted as the statement $\exists x \text{ CD-ACT}(x)$ which says that it is some unspecified CD act.

As it is getting apparent by now, language verbs get associated with compound CD structures. Another set of verbs, dealing with mental actions, also translate to causal relationships. These are words like *prevent*, *instigate*, *hurt*, *comfort*, *advise* and *threaten*. We look at the representation of the last one. Conceptually the act of threatening is to communicate by some means to someone that if they do some particular thing the response by the threatener will not be pleasant for them. Here, in Figure 13.13, x is threatening y.

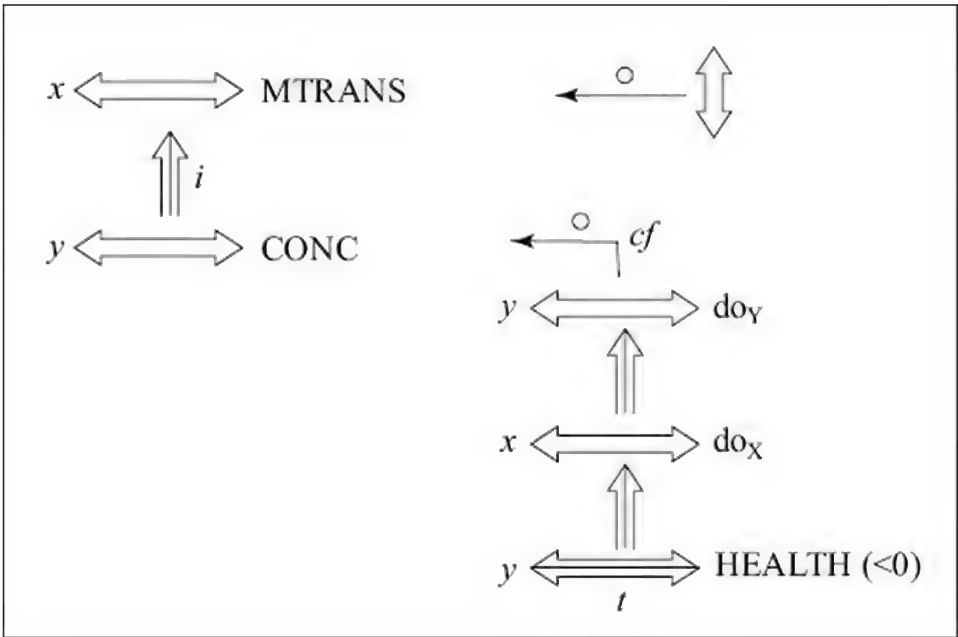


FIGURE 13.13 Threatening is a communication act in which a person conveys dire consequences of doing something to another person.

Words like *love*, *hate* and *like* are also not conceptual actions, but causal connections. When Sheeri says that she loves Farhad, she is saying that when she thinks of him she goes into a state of being pleased or happy. When Abasi says he likes ice cream, he means that he conceptualizes that if he were to eat ice cream then that would result in him going to a pleased state. This may be represented as in Figure 13.14.

The act of throwing something can be represented as PTRANSing it in the air using the instrumental act of PROPEL and GRASP. So if we said that “*Adriana threw a pencil towards Ayumu*”, then we could represent it as shown in Figure 13.15.

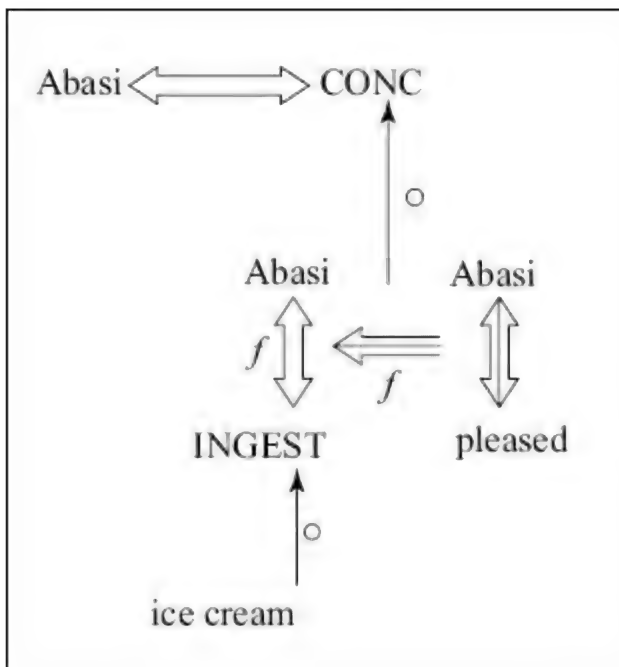


FIGURE 13.14 Liking ice cream is thinking that one will be pleased on eating it.

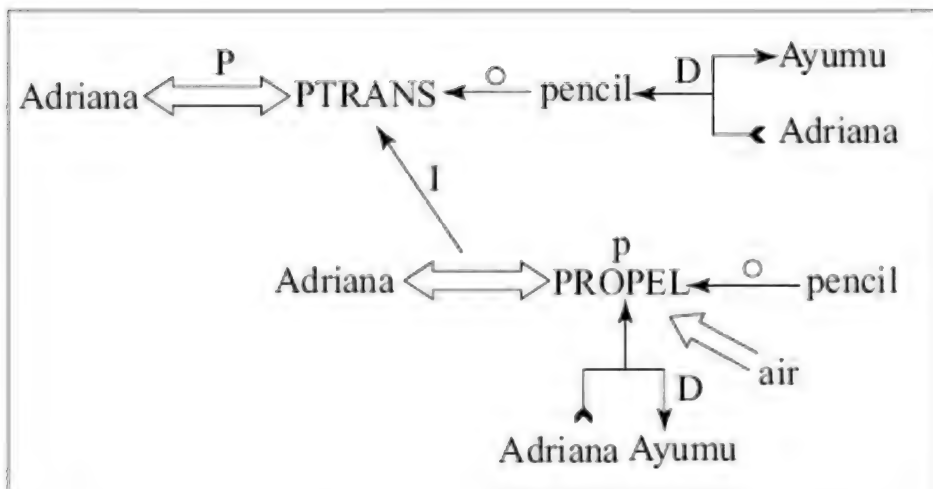


FIGURE 13.15 Throwing something is PROPELing it through the air.

The above formula/diagram represents the act of throwing the pencil in the direction of Ayumu. This could be part of a larger conceptualization. If Adriana's intention was to *give* the pencil to Ayumu then the above would be the instrumental act for an ATRANS action. If on the other hand, the pencil was thrown with the objective of hitting Ayumu then the above would be related causally to a PROPEL act in which the pencil comes into contact and applies force to Ayumu.

One may need to clarify on instruments as well. Linguistic instruments like forks, spoons, hammers, shovels, and mobile phones are often objects of conceptual actions. For example, if we said that Abasi ate the ice cream with a spoon, then the spoon becomes an object of a PROPEL and PTRANS act which

are instrumental to INGESTing. Likewise, if we had said that “*Adora killed the cockroaches with insecticide*” then while the insecticide is linguistically an instrument of the killing act, conceptually it is only an object of an action that is causally related to the state change event of the cockroaches being killed.

13.7 Conceptual Analysis

If knowledge representation and reasoning is the “cognitive” activity that happens in support of an agent acting intelligently, how does an agent process language? Or for that matter information being acquired from external sources in any other form. Obviously, the goal would be to transform the information into its internal representation in order to be able to reason with it.



FIGURE 13.16 Software agents have to make sense of information coming via the keyboard or some other medium.

Consider the design of a game playing agent (see Figure 13.16). The agent has some internal representation of “the world” along with representation of the moves one can make. The opponent makes moves via the keyboard or a joystick or a mouse, or even a process if the opponent is a program. The task of the agent is to decipher the signals by mapping them to the representation of states and moves. Or consider a futuristic scenario in which a spy is being debriefed by a computer program. The spy narrates her story and the program has to understand it, and ask meaningful questions to fill any gaps that it might see. The problem of understanding is to map the incoming information into the internal representation. We focus on the task of natural language understanding.

Does understanding happen in a bottom up fashion or in a top down fashion? By the former we mean that the agent makes sense of each word first, uses (grammar) rules to form phrases, and sentences. And finally builds the big picture by combining all “parsed” information in some context.

By a top down process we mean working with a (hypothesized) structure in which gaps have to be filled. If we can somehow create a scaffolding of the final story, then all we need to put in are the details. Some of this scaffolding will come from world knowledge that we have. For example watching the last few minutes of soccer match and knowing the score we *expect* the team that is trailing to do something dramatic, and when they do we “know” why they did that. Or walking into a James Bond or an Indiana Jones movie we expect to see some pretty outlandish action, and devour the film with a suitably receptive frame of mind. Or listening to a *khayal* in Indian classical music we know the structure of the composition and expect the notes and the rhythm to build up to a crescendo. When

we have figured out what the *raga* is, we no longer need to make sense of the individual notes in a bottom up fashion. Rather we bring to fore our preconceived patterns in that *raga* and expect the notes to match them. It is top down reasoning that is responsible for humans seeing a face amongst the clouds, and it what predisposes us to a particular interpretation of an ambiguous optical image, or a sentence.

Language understanding is also influenced by expectations of the listener. The expectations originate from various sources. We may know the speaker to be a terrible bore, and expect a long and painful narrative about his exploits. Or we may know the situation and generate expectations from that. For example, if two boys are fighting over some marbles, we expect to hear accounts of how the game was played and who won the marbles or who cheated. Some of these expectations that are generated by world knowledge we will look at in the next chapter, when we look at knowledge structures. For the moment we look at expectations that are generated by the individual words that appear in a sentence.

Some words always occur together and we can generate expectations at the lexical level. For example phrases like “inexorably squeezed”, “ulterior motive”, “corrupt officials”, “hopping mad”, “thank you” and “good luck” occur frequently enough to predict the second word. But the kind of expectations we are interested in here are those that are generated by the *meaning* to the words seen so far. For example if we hear, “Ramesh ate a ...” we expect to hear of some food type object. Once we know that an act of INGESTING is being spoken about, we know that there must be an animate eater and some edible object. This kind of semantic knowledge would allow us to make sense of grammatically ill formed sentences as well, suggesting that syntactic parsing is not necessarily a precursor to semantics. For example if a baby says “Amit eat apple” or even “apple Amit eat” then it is not a major handicap that the sentence is not well formed. Situation knowledge may in fact help us understand the sentence easily. One could figure out whether the child Amit *wants* to eat an apple, or whether it *has* eaten one, or whether some third person called Amit has eaten one. One of the theories that Schank put forward is that pragmatic and semantic knowledge in fact helps the process of language understanding, and do not follow a parsing stage.

This is particularly important because natural languages by themselves tend to be richly ambiguous. Words can assume different syntactic categories leading to different valid syntactic structures of the sentences they occur in.

Perhaps this is best illustrated by the celebrated example “*Time flies like an arrow*” (Kuno and Oettinger, 1962). Most human beings arrive at one (implicit) parse tree without any hesitation. But a program written on the IBM 7090 computer in the early sixties armed with 3000 grammatical rules, and an unprejudiced world view, could not decide between competing meanings and parse trees. Is *time* a verb referring to the act of timing flies in which case one is advised to follow the procedure for timing arrows, or is it an adjective describing a particular kind of flies that like an arrow? When we hear the sentence we move quickly into a top down mode choosing time to be a noun and building the rest of the meaning around it. Obviously, this has the advantage of speed, and is an approach that natural language understanding programs should look at too.

This is not to say that syntax has no role to play. If one were to read about the Charlie Chaplin film, *The Gold Rush*, in which “the tramp ate the shoe”, then heuristics that will look for an edible object to fill the object slot in the INGEST conceptualization will fail. It is only the grammar rules that force us to accept the fact that the shoe was in fact eaten. Or one could fill in the slot as a last resort when the sentence is finished, and perhaps flag a warning. Another example that

illustrates the utility of grammar rules is the sentence “*I saw the Grand Canyon flying into New York*” (Schank, 1973). After you have sorted out the ambiguity of who or what is flying, one still needs to establish a connection between the two constituent conceptualizations—I flying into New York, and, I seeing the Grand Canyon. A language specific rule says that the former marks the time for the latter. The entire CD structure is depicted in Figure 13.17.

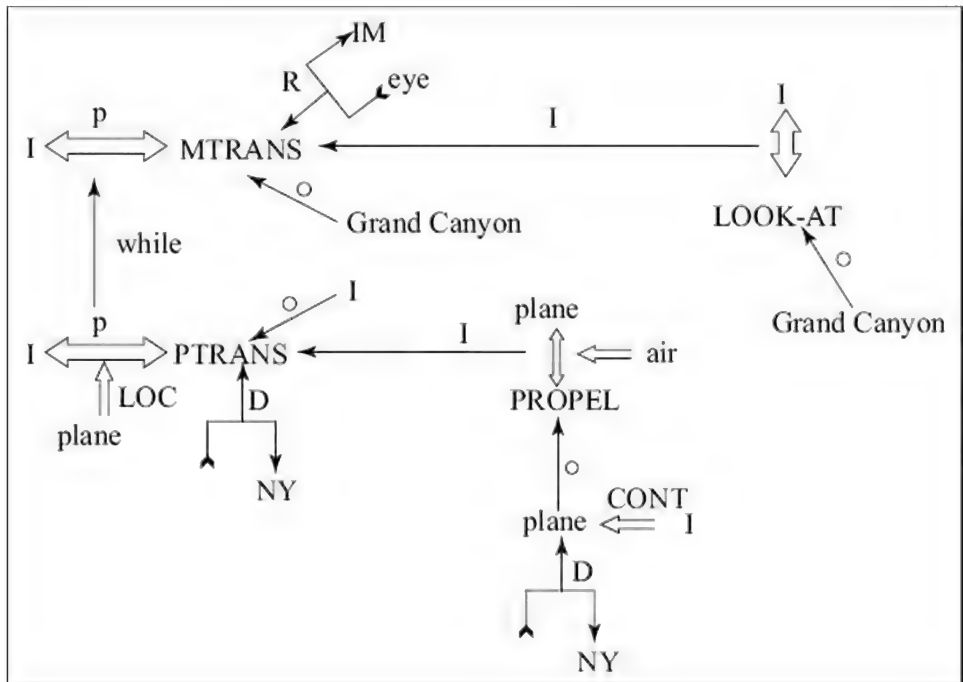


FIGURE 13.17 An English specific rule says that one action (seeing) can happen *while* another event is happening.

Observe that in our model it is the plane that is PROPELing itself in the air. This, flying *in a plane*, is one of the senses that one can accept the *word flying* in, apart from actually flying a plane, or flying by oneself which only birds and insects (or Superman) can do.

Conceptual analysis or semantic parsing is a two stage process. First the skeleton of the conceptual structure has to be hypothesized. This is done by retrieving structures from a dictionary that stores CD structures against each linguistic verb. In the second phase the empty slots in the hypothesized structures have to be filled in with information extracted from the processed utterances.

To bring to fore the semantic knowledge while processing words, one needs to construct an appropriate dictionary or lexicon. Apart from storing syntactic categories, it should also store the different word senses in the form of CD structures with each entry.

13.7.1 Semantic Rules in a Lexicon

The following description of a semantic parser is based on the work that came out of the Yale group (Schank et al, 1973; 1975), (Riesbeck, 1975), and (Birnbaum and Selfridge, 1981). The main resource used by the parser is a semantic lexicon that maintains information of semantic relations in the forms of rules for each word.

A word with more than one sense would have more than one entry, and it would be the job of the parser to select the right one. This could be done by search, employing semantic matches to choose between options where possible.

The parser stores partly filled conceptualizations in its immediate memory called *C-list*. When a word is read, its entry from the lexicon is retrieved and added to the *C-list*. Along with its *CD* structure, the rules stored with the entry are also retrieved and stored in its rule memory called *R-list*. Before reading the next word the parser looks at all rules waiting in the *R-list* for any that are applicable, and executes them.

The program written by Riesbeck called *ELI* (English Language Interpreter) activated instances of rules to establish semantic connections. The activated rules were called REQUESTS. The entry in the semantic lexicon for the word “ate” would contain one REQUEST with precondition *T* (no condition) and which would activate two other requests. The entry for “ate” looks like,

```
ate :   category -verb.  
       forms -eat. eaten  
       REQUEST -  
         TEST:      T  
         ACTIONS:   Add the structure  
                    (INGEST ACTOR (NIL) OBJECT (NIL) TIME (PAST))  
                    to the C-List  
  
                    Activate the request  
                    REQUEST -  
                    TEST: Can you find a human on the C-list preceding the INGEST  
                          structure?  
                    ACTIONS: Put it in the ACTOR slot of INGEST  
  
                    Activate the request  
                    REQUEST -  
                    TEST: Can you find an edible object on the C-list following the  
                          INGEST structure?  
                    ACTIONS: Put it in the OBJECT slot of INGEST
```

The single request stored in the entry for “ate” would add the INGEST structure, and also the rules needed to find fillers for the slots in the structure with semantically matching concepts.

Requests when nominals or PPs are read would be to add the appropriate structures to the *C-list*. For example while reading the sentence “Amit ate an apple” first the single request that would be activated is

```
REQUEST -  
  TEST:      T  
  ACTIONS:   Add the structure  
              (PP CLASS (HUMAN) NAME (Amit))  
              to the C-List
```

This would get activated as soon as “Amit” is read, and the structure (PP CLASS (HUMAN) NAME (Amit)) would be added to the *C-list*. When the word “ate” is read next, the structure for INGEST would be added, and the two requests added to *R-list*. The parser would then look for the “human” PP structure in the *C-list* and move it to the ACTOR slot. The next word the parser would read is “an”.

The following request stored with the word “an” is added to the *R-list*,

REQUEST -

TEST: Has a new structure been added to the end of the C-list?

ACTIONS: Mark it as an indefinite reference.

When it is added it does not find any structure in the *C-list* and the parser goes on to read the word “apple”,

REQUEST -

TEST: T

ACTIONS: Add the structure
(PP CLASS (FOOD) NAME (apple))
to the C-List

This now activates the rule for “an” which modifies the structure to
(PP CLASS (FOOD) NAME (apple) REF (INDEF))

This modified structure is now taken up by the other request for INGEST that is looking for food type object, and it inserts it into the OBJECT slot of INGEST. The final conceptualization is,

(INGEST ACTOR ((PP CLASS (HUMAN) NAME (Amit)))

OBJECT ((PP CLASS (FOOD) NAME (apple) REF (INDEF))) TIME (PAST))

A simple algorithm to parse sentences is to read the words one by one, retrieve the set of rules associated with the word and add them to the *R-list*, and then apply the rules that are activated. An outline is depicted in Figure 13.8 where we have not considered multiple word senses. If multiple word senses are allowed more sophisticated processing, like backtracking or least commitment strategy, would be needed. We assume that our algorithm retrieves the correct set of rules non-deterministically. We also assume that we have a function *Apply(request, cList)* that takes an applicable rule, tested by *RequestTest*, and makes the appropriate modification in the *cList*, and removes the *request* from the *rList*. In that sense *rList* is treated as a global list.

13.7.2 Case Markers

The reader would have noticed that the parsing of the above sentence made use of the order information typical of an active sentence in the English language. This is necessary because English is a language in which the roles of different participants of a sentence are indicated implicitly in the sentence structure. A sentence like “The apple was eaten by Amit” is an example of a passive sentence marked by the phrase “was eaten by” and would have a different set of requests associated with it.

Moreover, to justify our claim that the language understanding is not critically dependent upon syntax, the parsing algorithm would have to find a best fit semantic role in utterances like “eat apple Amit” overruling the position requirements. At other times when a semantic match does not occur, for example in “the tramp ate the shoe”, one has to rely on the syntactic order to fill “the shoe” into the OBJECT slot of INGEST. Syntax also plays a key role in making sense of sentences like “man bites dog” or “man swallows snake” where the semantic roles

are not clear. This could possibly be handled in the above algorithm by checking for any pending requests before exiting. A secondary procedure could then relax the semantic matching criteria for activating the pending requests.

```
SimpleSemanticParser(sentence)
1 rList ← ()
2 cList ← ()
3 while not Null(sentence)
4   do nextWord ← Head(sentence)
5       requests ← RetrieveFromDictionary(nextWord)
6       rList ← Append(requests, rList)
7       cList ← ApplyRequests(cList, rList)
8 return cList

ApplyRequests(cList, rList)
1 while not Null(rList)
2   if RequestTest(Head(rList)) = TRUE
3     then return ApplyRequests(Apply(Head(rList), cList),
                                Rest(rList))
4   else return ApplyRequests(cList, Rest(rList))
```

FIGURE 13.18 A simple semantic parser retrieves rules associated with words and applies them to partial conceptualizations stored in the *cList*. We assume the function *RetrieveFromDictionary* that retrieves the relevant rules nondeterministically. We also assume a function *Apply* that applies a request to the *cList*.

Many languages in the Indian subcontinent on the other hand employ *explicit* case markers. Many of these languages are based on the grammar given by Panini around the 4th century B.C. (Vasu, 1962) and recent work in linguistics has drawn upon it (Bharati and Sangal, 1990), (Bharati et al, 2004), (Huet et al, 2009).

The Paninian framework is interesting because it addresses the semantic issues as well. The grammar explicitly focuses on actions, the actor, and other objects in relation to the action. In the Panini grammar syntactic constructs called the *vibhaktis* are used to explicitly mark the semantic role of each constituent via *karakas* or semantic role indicators. These *karakas* manifest themselves as morphological inflexions on the base word in some languages, including Sanskrit, while they occur as separate markers in others like Hindi.

Panini specified six ways in which constituents of a sentence can relate to the verb. These are the following (Vaidya et al, 2009),

- | | |
|-------------------------|--|
| k1: <i>karta</i> : | central to the action of the verb |
| k2: <i>karma</i> : | the one most desired by the <i>karta</i> |
| k3: <i>karana</i> : | instrument which is essential for the action to take place |
| k4: <i>sampradaan</i> : | recipient of the action |
| k5: <i>apaadaan</i> : | movement away from a source |
| k7: <i>adhikarana</i> : | location of the action |

Of these the first two must be present for a sentence to be complete (and grammatically correct) and the others are optional. Thus *karakas* mark the role of a constituent in a conceptualization, and often have a direct relation to the role markers or *vibhaktis*. In some sense the *vibhaktis* are like indices to the kind of rules that *ELI* would employ to find proper fillers for a slot in a conceptualization. Languages with explicit role markers are often free word order languages since the grammar does not impose an ordering on the clauses.

13.7.3 Word Phrases

Words in a sentence seldom stand alone for something. Usually they embellish or are embellished by other words and together they form a phrase that stands for something. Linguistics identifies such groups as verb phrases or noun phrases. Since phrases introduce a hierarchical structure into the sentence, and also the underlying conceptualization, they have to be dealt with in a special manner. For the top level sentence it would be nice if the phrase as a whole was available. But reading or processing words in a sentence happens sequentially, and understanding or parsing the phrase is a sub-task in itself. A bottom up approach would incrementally move up the hierarchical structure, putting together smaller components to form larger ones.

Parsing a sentence has to be a judicious mix of top down and bottom up processing. In the example we saw of *ELI*, this was achieved to some extent by storing partially built constituents in the *C-list*. But one still needs to be careful as demonstrated in the processing of noun phrases in the heuristics proposed by Gershman (1977). The basic idea that Gershman proposed was that the top down processing of the sentence should be suspended when the parser is looking at a noun, and should resume only when the end of the noun phrase occurs. The control algorithm needs to switch between the top down predictive and the bottom up agglomerative modes of processing.

A well known example is the phrase “cat food can cover”. Observe that processing the sentences left to right one will end up “recognizing” the following, before having to revise the structure. For example, each of the following would have been a valid conceptualization, but for the words that follow it,

- He picked up the cat. (The object is the cat)
- He picked up the cat food. (The object is the food)
- He picked up the cat food can. (The object is the can)
- He picked up the cat food can cover. (The object is the cover)

Human beings often run into difficulties incorporating the bottom up mode needed for processing such sentences and end up backtracking and discarding the partial structures they have built while listening to or reading the sentence. Researchers have named such “difficult” sentences as *garden path sentences*, since they initially lead you astray from the final conceptualization (see for example (Ferreira et al, 2001)).

Some examples of such well known sentences are,

- While Anna dressed the baby spit up on the bed.
- Mary gave the child the dog bit a band aid.
- The old man the boat²⁶.
- The cotton clothing is made of grows in Mississippi.

What such sentences do is reinforce the idea that we humans predominantly do top down processing actively generating expectations along the way. When the expectations are not validated in garden path sentences we are forced to revise them. Such violation of expectations is often the basis of humour as well. While listening to a joke we are often led down a garden path, before seeing the funnier side. For example the following sentence is attributed to the comedian and film star Groucho Marx, “Time flies like an arrow. Fruit flies like a banana”. Another example where we are led up a garden path as to the syntactic category of the word “flies” is the joke²⁷,

Question: "What has four wheels and flies?"
Answer: "a garbage truck"

13.7.4 Homographs and Word Senses

Natural languages are replete with words that mean different things (homographs or homonyms) or have different semantic senses in different sentences.

Homographs are words that have more than one concept (structures) associated with them. Examples of such words are plane, bank, terminal, mouse, fly, glass, and cricket. The following sentences contain two instances of a homograph with different senses.

1. Asiya *rose* to find a *rose* on her window sill.
2. It *does* not bode well for the ecology to kill *does*.
3. The *wind* blows hard on the roads that *wind* up the mountain.
4. A sitar in *sound* condition will produce a good *sound*.
5. The soldiers may *desert* the legion in the hot *desert*.

Retrieving such a word would mean having to choose from the competing structures. This could either be done by searching for the correct match, trying them one by one, or it could be done by bringing knowledge from different sources to the fore. In the following chapters we shall look at some of the approaches that exploit other forms of knowledge.

Prepositional words like *in*, *on*, *by*, *to* and *with* are often used to link up words in many different ways. Consider for example the use of *with* in the following sentences.

1. The bottles are filled with wine.
2. The bottles are filled with automatic machines.
3. Adora killed the cockroaches with insecticide.
4. Abasi ate the ice cream with gusto.
5. Abasi ate the ice cream with a spoon.
6. Their hearts were filled with pride (like the bottles filled with wine but not *quite*).
7. He shot the girl with the rifle.
8. He shot the girl with the boy.
9. She ate the ice cream with the boy.
10. He fought with his brother against the intruders.
11. He fought with his brother.
12. He went pale with fear.
13. She left the notebooks with me.
14. Ajinkya's brother used to live with him.
15. She helped the man with the broken arm.

Schank reported the following four conceptual realizations of the phrase "with PP". The *ELI* parser considers the senses in the given order and chooses the first one that fits.

1. PP is the object of the instrumental case. In sentence 3 above the insecticide is the object of some action that is an instrumental act for the state change verb *kill*.
2. PP is an additional actor of the conceptualization. Sentence 9 is an example.
3. PP is an attribute of PP immediately preceding it. In sentence 15 "with the broken arm" describes the man.
4. PP is an attribute of the actor of the conceptualization. We can think of "gusto" in sentence 4 as an attribute of Abasi, though perhaps it also

describes the action to some extent.

In addition we can find roles that the word “with” plays in the building of a conceptualization. In sentence 13 “with me” identifies ‘me’ as the recipient of the ATRANS action. Sentence 14 says that Ajinkya’s brother stays in Ajinkya’s house. In sentence 12 there is a causal connection between the state of being afraid and the state change of turning pale. Sentences 11 and 12 have the common phrase “with his brother” though the relation between the two in the respective conceptualizations is quite different.

Some of the sentences in the list above are inherently ambiguous in the sense that “with” is used in. In sentence 8 the boy could be an attribute of the girl (which could have been mentioned to distinguish between two girls on the scene) or it could have been like in sentence 9 where the boy is a co-actor. Likewise in sentence 7 the rifle could have been an instrument in the shooting act, or it could have been an attribute of the girl, while the shooting could even have been done with a camera. Even in sentence 5, the spoon could have been an attribute of the preceding PP (ice cream), for example in response to a question “Which ice cream did Abasi eat?”. Finally in some world it might be conceivable to imagine that each bottle in sentence 2 is filled with a nano machines that work automatically.

While word sense disambiguation can sometimes be done with information available within the sentence, and it is a matter of finding efficient ways of doing so, when a sentence is inherently ambiguous it is only information from its context that can help resolve the ambiguity.

One needs knowledge from all sources, syntax, semantics and pragmatics to be able to quickly make sense of natural language. Trying to order the three processes in some fixed manner is only likely to be futile. We shall look at language processing in more detail in Chapter 16.

13.7.5 Expectations: Other Sources

The semantic parser sketched above is based on the idea of meeting expectations generated by the meaning of the words seen so far. However, sentences in a language rarely occur in isolation. There are cues or expectations that arise from the context. This may be the topic of discussion for the sentence and may bring world knowledge to the fore. This may be the discourse itself. If the words in an utterance are part of a dialogue between agents, then the expectation of a complete conceptualization may itself be waived. This happens typically in a question answer session. If you ask a child her name the answer often consists of just the name, rather than a full sentence. Likewise if ask a question like “When did you get hurt?” you often expect a one word answer like “yesterday”.

The reason why we are able to deal effortlessly with such replies is because we do not expect them to form complete conceptualizations on their own. Rather we already have a partially filled conceptualization, and the answer supplies some of the missing pieces.

The context of utterance may also be a source of expectations. Waiting outside a dentist’s office we expect our name to be called and know that it is the summons into the chamber. Knowing that an election is taking place we have expectations of an announcement of the winners, of news of celebration, or of recriminations of unfairness. In the next chapter we look at how some of the knowledge about the world we have can be represented, and how it can come to fore to help us effectively deal with language and problems in an effective way.

13.8 Discussion

As human beings we are accustomed to ‘thinking in a language’. The philosopher Wittgenstein said that “*The limits of my language mean the limits of my world.*” (Wittgenstein, 1921). The Sapir–Whorf hypothesis says that the grammatical categories a human uses in language strongly influence how the person understands the world (Whorf, 1956). The most well known example of this is in Whorf’s study of the language of the Inuit people, who were thought to have numerous words for snow. Kenneth E. Iverson, the originator of the APL programming language, believed that the Sapir–Whorf hypothesis applied to computer languages. Noam Chomsky’s theory of linguistics is built around the notion of a *Universal Grammar* that all humans are innately born with (Chomsky, 1965). The key point however is that symbolic representation, or semiotics, appears to be the stepping stone to thought.

It is still not quite clear how to represent certain concepts like water, air, and other kinds of materials. Is one talking of the material itself in an abstract sense or is one referring to a specific occurrence of the material? How can we represent the notion of “water”? If one said “*Ainka drank a glass of water*” then we can represent the fact the glass contained something, but how does one define the semantics of the term “water” here. It is surely not a discrete “object” in the domain the term can map to. We tend somehow to think of water as fluid, flowing, something that takes the shape of its container, something that tends to settle down at the lowest level, and a type whose instance is not discrete. How do we represent the fact that some amount of this *kind of stuff* is what was contained in the glass, and that Ainka drank? Yet at the same time we do not have a problem with the sentence “*Ainka picked up a cube of ice*” because that is somehow a single discrete object. What does it mean to say “*Ayaka likes to drink water*”, or “*Accalia like to splash around in water*”, or “*Water is essential for life*”? These are issues that have to be solved before a machine can knowledgeably interact with us in our languages. The following example from Sowa (2000) gives us a clue:

Clyde is an elephant.
Elephant is a species.

Can we infer that “Clyde is a species”? No, because the word *elephant* in the second line is something that belongs to a type of thing called a “species”. It is an element or term here, a reified term, which belongs to the class “species”. This is similar to the fact that Clyde in the first sentence is a term that belongs to the class “elephant”. This means in some situations we have to treat a class (elephant) as a term which belongs to a higher level class. Linguistically, the use of the indefinite article “a” gives us a clue that we should utilize. We can then think of water too as a term which belongs to a class of fluids, and which has properties like taking the shape of the container, and something one can splash around in.

In this chapter, we have focused on variables and categories in an *abstract* sense. There is another aspect to knowledge representation, one that is concerned with efficiency, called *concrete* representation in (Charniak and McDermott, 1985). This is concerned with how one accesses the related knowledge without having to scan and unify a flat representation. It is concerned with the issue of designing structures for rapid access, so that reasoning does not have to rely on having to search for all the links that are needed in an argument. We will look at scripts, structures of goals and plans, and notion of ontologies and descriptions logics in more detail in the following chapter.



Exercises

- Convert the following definitions into Horn clause (logic programs).
 - $\forall x(\text{Mother}(x) = \exists y \text{ Mother}(x, y))$
 - $\forall x(\text{PrimeNumber}(x) = \neg \exists y(y \neq 1 \wedge y \neq x \wedge \text{Divides}(y, x)))$
 - $\forall x \forall y (\text{GrandParent}(x, y) \equiv \exists z (\text{Parent}(x, z) \wedge \text{Parent}(z, y)))$
 - $\forall x \forall y (\text{Mother}(x, y) = (\text{Parent}(x, y) \wedge \text{Female}(y)))$
 - $\forall x \forall y (\text{Ancestor}(x, y) = (\text{Parent}(x, y) \vee \exists z (\text{Parent}(x, z) \wedge \text{Ancestor}(z, y))))$
- Create a FOL representation scheme for describing the entities that are relevant to a University environment, like courses, people, departments, programs and so on, and relations between them. Rephrase the information in the form of RDF triples.
- Define the category of natural numbers,
 $NN(x)$ is true iff x belongs to the set of natural numbers.
Hint: Look up the Web for Peano Axioms
- Given a number N the von Neumann definition of natural numbers constructs the next number to be taking the union of all elements of N , and N itself. What would the union of all natural numbers correspond to? [*Hint:* Look up ordinal numbers].
- Write functions to convert measured values like,
feet and inches \rightarrow cm
miles and yards \rightarrow cm
kms and metres \rightarrow cm
- Write functions to add length measurements expressed in mixed units.
- How does one express the following in FOL? You may ignore tense information.
 - Anish ate an apple.
 - Anish ate the apple.
 - Arnav hit the boy with a stick.
 - Arnav hit a boy with the stick.

Would you represent “apple” as a term or a predicate for the above? How would you ensure that your representation is saying exactly what the sentence says? For example in (a) and how do you ensure your representation is true to the fact that Anish ate exactly one apple?
- What would a fuzzy membership function for the category “young adult” look like? What about “low salary”, “medium height”, or “too hot or too cold”?
- Given the following board position in Cross and Noughts, what was the first move made in the game? Assume that both the players are perfect (i.e. no losing move is made). Is there unique sequence of moves that with perfect play would lead to this position? (*Source:* http://www.geocities.com/joe_kisenwether/Retro.html)
- In the event calculus example, add the fluent $\text{satiated}(\text{Person}, \text{Time})$. What domain axioms does one need to add to be able to deduce that (after Nikhil has eaten) $\text{satiated}(\text{nikhil53}, t_7)$ is *true*?
- Modify the event calculus axioms EC1 to EC4 so that the effect of the event at time point t_1 is felt at a later time point t_2 . Apply the modified axioms to the example problem in the text.
- Given the *Spin*, *Load* and *Shoot*(x) actions and the fluents *Loaded* and

$Alive(x)$, assume that the ONLY effect of $Shoot(x)$ action is that $Alive(x)$ becomes false if $Loaded$ is true. The effect of $Load$ is that $Loaded$ becomes true. $Spin$ releases the fluent $Loaded$. Express the domain knowledge in Event Calculus. Given that following events take place at the times mentioned – (Load, 10), (Spin, 20), (Shoot(A), 30), (Shoot(B), 40), (Spin, 50), (Shoot(C), 60) – add the required formulas to the domain predicates that will entail that A and B are either both alive or both not alive at time 80. What can you say about state of C at time 80?

13. Define fluents and domain axioms to reason about the use of a telephone. The representation must be able to talk about the different states of the phone instrument (idle, ringing, dial tone, connected, engaged tone, disconnected and so on), the different actions (dial, pickup, putdown, and so on), and the relations between them (reference (Mueller, 2006)).

14. Express the following story (repeated from Exercise 15 in Chapter 12) using the Event Calculus and the CD theory predicates,

A father and his son were walking along the road when they met with an accident. The father died on the spot and the son was rushed to the hospital. When he was brought to the operating table the surgeon refused to operate upon him saying “I cannot operate upon this boy. He is my son”.

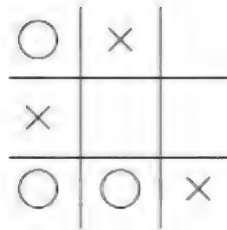


FIGURE 13.19 A retrograde analysis problem on Noughts and Crosses.

15. Map the following words/ phrases into relevant CD states—livid, hopping mad, unwell, relaxed, afraid, sleepy.
16. Create CD representations for the following sentences,
 - (a) Ainmere bought a book.
 - (b) Adriana threw the pencil to Ayumu.
 - (c) Adriana threw the pencil at Ayumu.
 - (d) The bottles are filled with wine.
 - (e) The bottles are filled with automatic machines.
 - (f) Adora killed the cockroaches with insecticide.
 - (g) Abasi ate the ice cream with gusto.
 - (h) Abasi ate the ice cream with a spoon.
 - (i) Their hearts were filled with pride.
 - (j) He shot the girl with the rifle.
 - (k) He shot the girl with the boy.
 - (l) She ate the ice cream with the boy.
 - (m) He fought with his brother against the intruders.
 - (n) He fought with his brother.
 - (o) He went pale with fear.
 - (p) She left the notebooks with me.
 - (q) Ajinkya's brother used to live with him.
 - (r) She helped the man with the broken arm.
17. Convert the following stories into CD like FOL representation.
 - (a) “Heidi told her grandfather that Clara was likely to come to their home.

Some time later, Clara came to their home. Heidi was very happy."

- (b) "Drona put down his weapons because he came to believe that his son was dead. This was because Yudhishtra told him that Aswathama was killed by Bhima."
 - (c) "Bush told Blair that Osama was a bad man and that is why Bush was going to bomb Afghanistan. This happened after the planes crashed into the WTC buildings"
18. In the English language one often finds the use of "of" in phrases like "six yards of Kanjeevaram silk", or "two litres of lemonade", or "a pint of beer". How would you write rules to handle such occurrences of "of" in a language processing system?

-
- ¹ *Mortal* and *Bright* are treated by FOL identically with *Man* and *Student*, though we might think of them as properties and not categories. Thus the unary predicate *Mortal* identifies the subset in the domain that is, or has the property of being mortal.
- ² Many people in the tropical areas in fact do not know snow first hand, and can only *try* and imagine its soft crunchiness and cold touch.
- ³ We use numeric suffixes to create unique constants in order to avoid the possibility of ambiguity in reference.
- ⁴ Or categories made up of collections of reified objects.
- ⁵ And are we too?
- ⁶ See <http://www.w3.org/RDF/> and <http://www.w3schools.com/rdf/default.asp>
- ⁷ See for example, Primer: Getting into RDF & Semantic Web using N3. <http://www.w3.org/2000/10/swap/Primer.html>
- ⁸ <http://dublincore.org/documents/dcmi-terms/>
- ⁹ <http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData>
- ¹⁰ <http://www.w3.org/blog/SWEO/>
- ¹¹ <http://creativecommons.org/>
- ¹² <http://raJweb.org/foaf/>
- ¹³ <http://thedatahub.org/group/lodcloud>
- ¹⁴ A clickable version is available at http://richard.cyganiak.de/2007/10/lod/lod-datasets_2010-09-22.htm
- ¹⁵ See <http://www.w3.org/TR/rdf-sparql-query/>
- ¹⁶ In the next chapter we will introduce the notion of classes that will stand for abstractions of individual elements. Classes will be defined as intensions based on their properties.
- ¹⁷ Apparently as a young student McCulloch was asked by Quaker Rufus Jones as to what he intended to become and what he planned to do, to which he replied "*I have no idea, but there is one question I would like to answer. What is a number, that a man may know it, and a man that may know a number?*". Jones, it seems, smiled and told him that he was going to be busy for the rest of his life.
- ¹⁸ Unless you are a bus driver in Chennai, in which case you often see it as green.
- ¹⁹ The squares of the chessboard are usually labeled a-h from left to right and 1-8 from bottom to top as seen by the white player. In using the values the values e8 and c6 we have *assumed* that white is sitting "below" the board.
- ²⁰ Instead of introducing a new predicate or fluent like "planToRide" one could introduce a generic EC predicate "*Try(event, time)*" as was done by Allen (1991) in the Event Calculus to accommodate actions. One could then introduce a rule that says $(Try(e, t) \wedge \text{preconditions of action}) \supset Happens(e, t)$.
- ²¹ Strictly speaking this should be done in two inference steps. In the first step the

Initiatesf..., ...) formula should be produced, and in the second step EC_1 is applied to actually initiate the fluent.

²² Provided we have the assertion: deflated(B) \equiv \neg inflated(B).

²³ The interested reader is referred to the many variations the character Cyrano de Bergererac presents to describe his nose (Rostand, 1897).

²⁴ <http://en.wikipedia.org/wiki/Natyashastra>

²⁵ but sometimes she does too.

²⁶ especially for Ernest Hemmingway readers

²⁷ http://en.wikipedia.org/wiki/Garden_path_sentence

Structured Knowledge Representations

Chapter 14

First Order Logic (FOL) provides us a language for describing the elements of a universe of discourse, and a mechanism for reasoning about the relations that exist in the universe. The representation is in the form of a set of sentences in the language of FOL, for which set the universe of discourse is a model. Not every true fact need be stated explicitly. Some facts can be deduced. The reasoning that we have seen in Chapter 12 is deductive reasoning, in which the statements that are entailed by the given Knowledge Base (KB) can be ascertained, and made explicit, by an inference engine.

To derive the entailed facts, inference engines need to select an appropriate sequence of rules. Every rule that has matching antecedents is a candidate for selection. An inference engine, or theorem prover, picks a rule and applies it, and repeats this process till the goal criterion is met. The task of finding a proof involves search. Most inference engines have built in search strategies. A goal directed backward search engine, like Prolog, picks the clauses in the order in which the user has stated them. The clauses contain both the rules and the facts. A forward search engine like OPS5, uses a conflict resolution strategy (see Chapter 5) for choosing the rule to apply.

Knowledge representation is concerned with what the problem solving agent knows about the world. The knowledge of an agent constitutes not only of the knowledge of the physical world, but also about happenings in the physical world. The world does exist, no doubt, because the knower is part of that world, but *for the knower, or the agent, the world exists only as the agent knows it*. One particular school of thought known as *Phenomenalism*, which is a kind of extreme *Empiricism*, holds the view that “*physical objects do not exist as things in themselves, but only as perceptual phenomena*”.¹

From a knowledge-representation perspective, the language of FOL is a mechanism to construct an ontological base for modelling the world as known by the agent. The semantics of FOL is defined with respect to a domain and a mapping from the language to the domain. Terms of the language are mapped to elements in the domain. The unary predicates of FOL define categories in the domain, and higher arity predicates define relations between elements of the domain. In this sense, FOL provides a

mechanism for *abstract* representation (Charniak and McDermott, 1985). Any *connections* between elements are buried in the logical formulas, mainly categories and rules, and have to be ferreted out by the inference engine by a process of search. The formulas exist in an abstract pool, which may be thought of as a flat representation with no structural information, except in Prolog formulas (programs) where they have a predefined order in which the inference engine looks at them. Figure 12.15 depicts some connections based on parent–child relationships that the formulas have implicit in them. If those connections could be made *concrete*, then the inference engine could traverse them directly. The algorithms in Chapter 12 do traverse such links, but only by finding them through a process of unification, and in the process of which they inspect all formulas, in some order, to determine whether there is a link.

The *search* for unifying components is what structured representations aim to circumvent.

There is motivation from human reasoning as well. We go about making decisions with varying degrees of effort in a dynamic world, facing and solving problem after problem. Parikh pointed that the truth functional semantics of logic is not sufficient. He illustrated the idea with vague predicates like “red”, and suggested that knowledge is useful when it leads to successful behaviour (Parikh, 1994). We jump to conclusions, we do top-down reasoning—employing whole structures of connected concepts. There is evidence from literature that we work with “chunks” of knowledge rather than a pool of formulas. Chess stalwarts from de Groot (1965) to Kasparov (2007) have emphatically said that it is the number of structured patterns in ones memory that distinguishes an expert from a novice in chess.

The key to the effective use of knowledge is the ability to get to the relevant pieces quickly. And that can be done if related pieces of knowledge are either chunked² together, or are reachable through explicit links. This chapter deals with ideas to combat the need to search for connections, and techniques to confine the processing to the “relevant” facts.

14.1 Hierarchies in the Domain

We begin with two features of knowledge representation. One is that *our* representations involve aggregations of simpler concepts into composite structures. We can model compound elements that are made up by assimilating and structuring basic elements. For example, we can think of human (as in a body) as a category, but we can also think of the different parts of the body as the right ear, or the left hand. Or we can think of a recipe for baking a cake, as made up of a number of smaller steps. We need to be able to handle such aggregations in which the parts are related to the whole and to other parts.

The second is the organization of categories themselves into

hierarchical layers of abstractions. For example, humans (bodies) are a kind of mammal (bodies) that are a kind of animal (by now, we only think of them as bodies and no minds!). Or a cake is a kind of baked food which is a kind of cooked food, and baking is a kind of cooking. Such kinds of *taxonomies* are part of *ontologies* in the computing community.

Both kinds of relations, between a part and the whole, and between different levels of abstraction can be captured in FOL, in the form of rules. But the problem is that arriving at conclusions by the process of search is computationally expensive at best, and intractable at worst. Instead, we need such *concrete* realizations of relations that finding related “facts” is done not by using search in a sea of formulas, but can be accessed more directly by following links.

Figure 14.1 illustrates the connections we seek to ossify. In one direction, an *aggregation hierarchy* relates components to composite elements. In an orthogonal direction, there exists an *abstraction hierarchy* that classifies elements into categories.

In the figure, categories are shown in shaded square boxes, and correspond to sets in the underlying domain. For example, the category “baby-girl” would refer to the set of elements in the universe that are baby girls. In the figure, each (large) oval represents kinds of things. Let us assume that the oval in the bottom left contains elements which—for the sake of illustration—are human beings. The oval containing the square boxes then is the set of categories. Other categories could be “schoolgirl”, “teenage-girl”, “teenage-boy”, “college-girl”, “mother”, “grandmother”, etc. Observe that the categories need not be distinct. They simply represent subsets of the universe. Further, categories may themselves be grouped into a higher level category. For example, the category “teenager” could contain the categories “teenage-girl” and “teenage-boy”, and the category “female” could contain the categories “schoolgirl”, “baby-girl”, “teenage-girl”, “mother” and “grandmother”. In fact, the categories at the second level correspond to supersets of the base elements, corresponding to the constituent categories. Yet we will add the categories themselves as *reified* elements to augment our domain so that we can reason directly about them. The lowest level in the abstraction hierarchy corresponds to sets of elements, or objects, in the domain.

The aggregation hierarchy defines other layers exemplified by the oval on the right. The elements of such a layer are *composite* elements. For example, an element at this layer might be a “family”. A family may be thought of as another reified element, which in *reality* is made up of its parts. A family in turn may be part of a larger unit, say a clan, which in turn may be part of a tribe, a community, a nation, and so on. Each of these levels could have abstractions of their own, giving rise to a complex network of kinds of things, when we add elements other than humans to our universe, both material and nonmaterial. Extreme examples are alluded to by noun phrases like “the market”, “the ecosystem”, “a spiral galaxy”, “the diaspora”, “the clergy”, “air pressure” and “the umwelt”.

These terms clearly stand for something that is an abstraction of aggregation at some level.

Both, the *abstraction* hierarchy and the *aggregation* hierarchy, are forms of reification in which we create symbols that *stand for something we have imagined*.

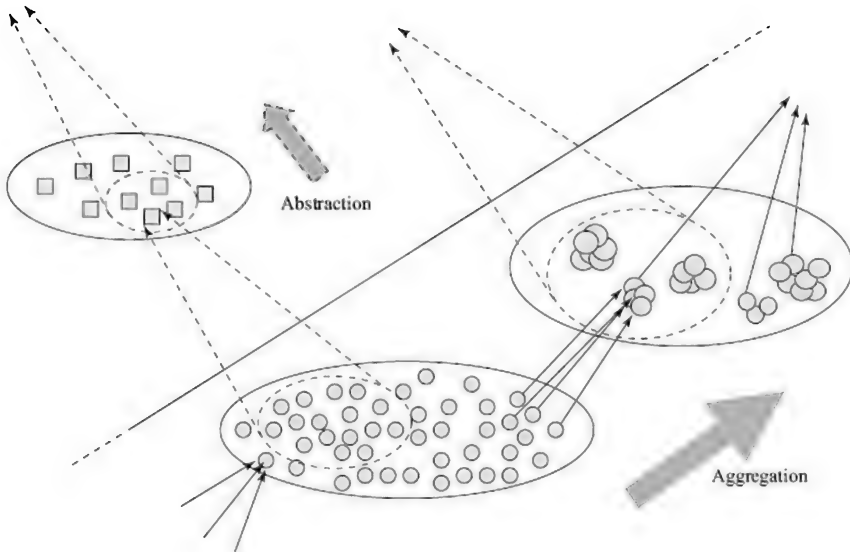


FIGURE 14.1 The abstraction and aggregation hierarchies.

Observe that while we have depicted a *lowermost layer* in the abstraction hierarchy, we have not done so for the aggregation hierarchy. The base in the abstraction hierarchy corresponds to elements (composite or otherwise) that “exist” in the domain. These are shown in shaded circles or their aggregations.

The base of the aggregation hierarchy likewise should correspond to the *smallest* level of existence, where the element is *atomic* in the sense of being indivisible. If we could define this *base* then the elements in that set would be the real domain, and everything else would be composite. However, the definition of what is the base or the domain is really a matter of choice, as far as *representation and reasoning* is concerned. It is the problem solver who has to decide as to what is the basic (atomic) unit in the domain. Fundamentally, we still do not know what is, if there is, a basic unit of matter from which other stuff is composed. A similar difficulty arises if we talk about ideas, properties, or events (remember Zeno’s paradox on motion). The notion of an atom was postulated in the sixth century bc as an argument against infinite divisibility of matter (Keith, 1921), (Gangopadhyaya, 1981), and adopted by John Dalton in the early nineteenth century, but that too turned out to be a composite structure.³

Nevertheless, a considerable amount of representation and reasoning

happens at different levels in the aggregation hierarchy. Physicists talk about strings and neutrinos, while chemists are content to deal with the atoms and molecules. Biologists and botanists work with genes and cells, while the physician thinks in terms of bacteria and organs of the human body. For sociologists, psychologists, anthropologists and economists, humans are the basic units. Ecologists and environmentalists are concerned with processes that impact the well being of the planet, while astronomers and astrophysicists consider the earth as a speck in the vast universe. A good feel of this hierarchy is given in the book based on the movie *Powers of Ten* (Morrison et al, 1985).

Figure 14.2 gives a concrete illustration of the two kinds of hierarchies over the set of people, with six particular individuals that were part of the famous Italian team that ruled the world of competitive bridge in the early latter half of the twentieth century.

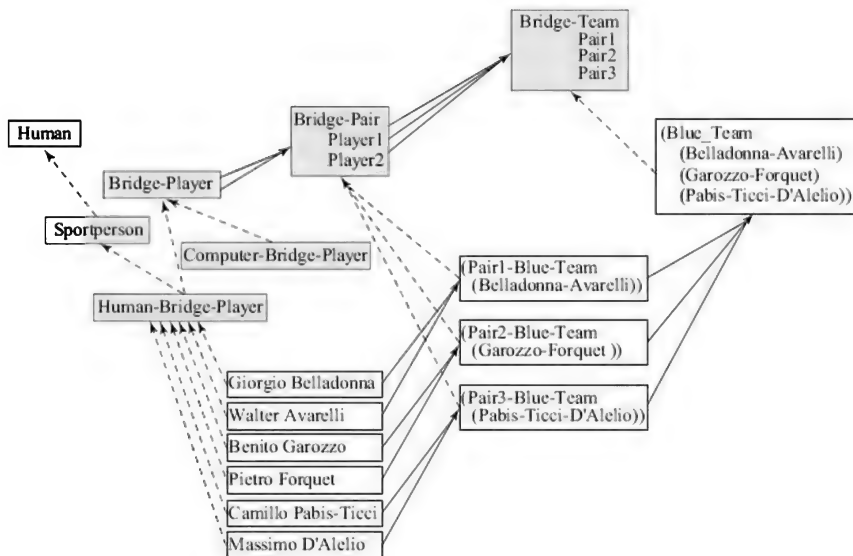


FIGURE 14.2 An example of the aggregation and abstraction relations. A bridge pair has two bridge players, and a team has three pairs. The team featured here is the famous Italian Blue Team.

Observe that the only “concrete” elements here are the six people. While the aggregations, the three pairs and the team, do exist on the ground, they involve the same six players.

These are not the only relations that exist. There are others which are more specific to different “nodes” in a wider *semantic net*, and are often between different kinds of things. For example, we have not represented bridge as a game. If we do so then we can define a bridge player as one who plays the game. All kinds of relations can be defined and captured. The game of bridge, for example, will have its own internal details, and the players who play it will employ a knowledge of patterns, strategies

and actions that are specific to the game. A complete representation of all knowledge in this sense seems difficult, if not impossible, either for man or machine. We just strive to know more and more.⁴

14.2 The Schema

An intelligent agent has to reason about the world in which it exists.

Fundamentally, the world is made up of some sub-atomic particles that obey laws that we are still trying to decipher, and whose movement results in the billions and billions of events that happen around us. Just consider the bacteria, good and bad, that inhabit a human body. Clearly, we cannot reason at the minutest level of detail. Instead we build, chunk, or group smaller elements into more manageable entities (aggregation), and create their abstractions, and (generalized) relations among the abstractions.

The aggregations we reason with exist in human-defined boundaries. We can most easily reason with lengths in the range of millimetres to kilometres, and have to stretch our imagination to venture outside this range. Our sensory organs are also attuned to certain ranges. The rods and cones in our eyes respond to the “visible” band in the spectrum of electromagnetic frequency, and our ears are only sensitive to an “audible” range of sound waves. Our sense of time is defined by seconds, minutes, hours, days, years and centuries.

The concepts (aggregations and abstractions) that we define, all lie within our perceptual boundaries. We see events that happen in seconds or minutes, and we perceive as *static* that change that takes longer or shorter. Air molecules are banging against our faces, electrons are whirring around the atomic nuclei, and the wings of the honeybee are fluttering. On the television screen, the computer monitor or a cinema screen change happens much faster than we can perceive; hundreds or thousands of times a minute. We only see the (illusion of the) smoothly changing image in *our* timescale, seeing a person’s lips moving or a football flying towards the goal. Beyond the other end of our perceptible time scale, the physical objects that we are and see around us are in fact undergoing change (beneath our very noses). The bud is blooming into a flower, the grass is growing, the glacier is moving, our hair is growing, the stars are moving across the sky, and night is beginning to change into dawn. But we do not see the change *happening*. Some change is far too slow for us to notice, for example the wood our table is made of is slowly oxidizing, or the rotation of the earth is slowing down, or our sun is expanding into becoming a red giant. Sometimes we simply refuse to acknowledge change, like global warming, even when others are frantically drawing our attention towards it.

The conceptual world we create in our minds is essentially a manageable knowledge representation strategy in which the number of “things” we deal with and reason about is comparatively small. A key

component that makes this possible is the notion of the *schema*.

The schema holds together and imposes a structure on a collection of simpler constituents.

The notion of schema has evolved during the quest for understanding human thinking, and is seen as something that is instrumental in assimilation and aggregation of similar experiences. The word schema derives from the Greek word *σχημα* which means “form”, “shape” or “figure” (see Table 12.1) and “indicates the essential commonality of a broad category”. It is a mechanism that facilitates generalization, synthesis, storage and retrieval of similar experiences (Marshall, 1995).

Like many ideas in knowledge representation, the notion of schema first appeared in philosophy. In Platos’s *Dialogues*, it appears to signify not only *form* and *shape* but also *abstraction* and *generalization*. In his *Metaphysics*, Aristotle relates the schema to his concept of *categories*, and thus to the *essence* of things. Schemas, according to Aristotle, facilitate the recognition and understanding of basic properties. The 18th-century German philosopher, Immanuel Kant, takes up the notion of schema in his *Critique of Pure Reason* (Kant, 1781). *A priori intuitions and concepts*, the term Kant uses for schemas, provide us with the framework for our *a posteriori* knowledge. “Things as they are “in themselves”—the thing in itself or *das Ding an sich*—are unknowable. For something to become an object of knowledge, it must be experienced, and experience is structured by our minds—both space and time as the forms of our intuition or perception, and the unifying, structuring activity of our concepts. These aspects of mind turn things-in-themselves into the world of experience. We are never passive observers or knowers.”⁵ This finds resonance with the modern view that we actively construct the worlds that we perceive.

The baton of studying cognitive schema was then taken up by psychologists. The two most prominent amongst them were Frederic Bartlett (1886–1969) and Jean Piaget (1896–1980). Bartlett, one of the earliest experimental psychologists, was interested in *how* people remember things and *what* they remember. His method of study was to make people listen to stories that, while looking normal on the surface, had some unrelated or illogical components. He hypothesized that what people remembered of the stories was what had matched the schemas—containing summaries of familiar stories and situations— that people carried in the heads. One of his celebrated stories called “The War of the Ghosts”⁶ (Bartlett, 1961) revealed that people tended to misunderstand the story and distort the parts that were misunderstood into something more familiar. Bartlett said that memory was composed by active organization and reconstruction of events, which were stored in an organized fashion rather than as individual episodes. Bartlett’s idea that schemas are used to *organize* data and are used to *anticipate* what will be heard next was taken by Schank and Abelson in their notion of *scripts* described below in Section 14.5.

Jean Piaget was more interested in learning. In particular, he was

interested in discovering how children learn the concepts of space, time, logic and mathematics. His studies have had lasting impact on computer science and artificial intelligence. Seymour Papert used his theories for devising the Logo programming language in 1967 (Papert, 1980) (see also Harvey, (1997)). On the issue of how schemas develop, Piaget postulated that they develop only for those situations that happen often. Like Bartlett, he too assumed that people tend to *squeeze in* the situation into a known schema, a process that he called *assimilation*. But if that is not easily possible, then the schema has to adapt to the new situation by a process of *accommodation*. Over a period of time, schemas become more generalized and also new ones emerge to cater to different situations. Like Kant, Piaget also believed in the active role the mind plays in perception. According to him, individuals are not passive creatures acted upon by the environment, assimilating experiences as they occur. Rather, individuals actively *construct* their own perceptions, assimilating new experiences and accommodating the schemas when needed. "*The schema, both structures our experience and is structured by it*" (Mandler, 1985).

It provides a framework which is useful in understanding related pieces of information being received by an agent. Because a schema holds together its constituents in a structured manner, when it is retrieved from memory, the entire structure with its relations is retrieved, and can help in coherently understanding what is going on. The work on Frames and Scripts described illustrates these features.

For both Bartlett and Piaget, schemas emerge when individuals strive to understand the world, and are direct consequences of their experiences.

A schema is what helps us organize and structure data. The data may pertain to objects in the world or to events. We learn and acquire schemas both through experience and by being taught. One of the first schemas that a child acquires is that of the human body, as it is repeatedly asked to point to its nose and eyes and lips and hands and feet. As it grows, the child acquires the notions of outings, shopping, rules of games, and how to tie shoelaces. For each of these, it develops some kind of internal prototype and learns to recognize and categorize things, as well as employ schemas for problem solving. Recall the way a child learns addition of numbers, with the procedure or algorithm of adding digits by "table lookup" and taking a carry over to the left and so on, and how it acquires more complex schemas to solve simple algebraic equations with one variable, to the method of bisecting an angle.

In the second half of the twentieth century, artificial intelligence researchers had begun to write computer programs to do all kinds of interesting things. Writing programs also became a means of testing theories of cognition, and it became imperative to put down the ideas into a concrete enough form to be expressed as programs and data. We shall look at three significant threads in the development of the schema in the works of David Rumelhart, Marvin Minsky, and Roger Schank.

Of the three, only Rumelhart continued to use the term schema, though his earlier work on story grammars tends to treat the schema as a somewhat vague concept in the background, focusing more on the set of rules to achieve the desired effect (Marshall, 1995). His later work along with Andrew Ortony (Rumelhart and Ortony, 1977) proposed the following characteristics of the schema,

1. *Schemas are data structures for representing the generic concepts stored in memory.*
2. *Schemas are defined for objects, situations, events, actions, and sequences of events and actions.*
3. *Schemas are not atomic but composite structures, containing networks of constituents and specified relations among them.*
4. *Schemas are like stereotypes of the underlying concepts.*

14.3 Frames

The genesis of modern *object-oriented programming* is in the notion of *frames* elaborated by Marvin Minsky (1974). The central idea in both is to pull together data pertaining to a composite conceptual entity in a meaningful fashion into a structure, so that its relations with the different constituents become obvious.

A frame system is a mechanism for chunking.

Frames also provide a mechanism for relating the chunked data (frames) to other chunked data (frames). One can capture the relations between two frames that are at a different level of abstraction. And one can capture the relation when one frame is a constituent of a larger frame. That is, one can capture both the abstraction hierarchy and the aggregation hierarchy shown in Figure 14.1.

A frame is defined by a name and a set of slot-filler pairs. Each slot-filler pair is made up of a slot (or property) name and a filler (or value). It is understood that the slot is a property of the frame, and the filler is the value of that property.

```
(frame-name
  <slot-name-1 filler-1>
  <slot-name-2 filler-2>
  ...)
```

The value that the filler can take can be an atom, another frame, a set, or a conditional procedure. An atom is a value that stands by itself, like a name or a number. An atom may be thought of as a frame with no slots, and only a name. A set has frames as its members (the frames could be atomic frames). A procedure may be attached to a slot to facilitate and control the flow of information. The trigger condition for the procedure decides when the procedure should be executed.

The sets below and to the right of the solid line in Figure 14.1 are sets

of *concrete* objects, in the sense that they *exist in the domain*, either individually or as composite objects⁷. An arrow *along* this dimension represents a *part-of* or *aggregation* relation, and is realized by placing the name of the constituent object (frame) in the filler slot of the composite object (frame), either individually or as a set. For example, we may represent the following,

```
(earth-system
  ...
  <:Planet earth>
  <:Satellite moon>
  ... )

(blue-team
  ...
  <:Pairs {blue-team-pair1, blue-team-pair2, blue-team-pair3}>
  ... )

(blue-team-pair1
  ...
  <:Players {georgio-belladonna, walter-avarelli} >
  ... )
```

We have used the notational convention of writing the slot name capitalized and prefixing it with the colon. We will also denote concrete elements (frames) in lower case letters. We expect the two constituent values of *Players* of the third example to be in turn frames that contain more information about the two persons. This is in fact universally true of fillers, keeping in mind that we treat atoms as frames with no additional information, apart from the name of the atom.

The sets above and on the left-hand side of the solid line in Figure 14.1 contain *abstract* objects that *stand for sets* of objects, abstract or concrete. An abstract frame can be thought of an abstraction of a *kind of frame*, which specifies all the properties common to the objects in the set it stands for. The direction orthogonal to the aggregation direction is the *abstraction* direction. Edges between objects along this dimension represent the abstraction relation. In the frame system, this edge is represented as a slot whose value is another non-atomic abstract frame, and that says that the current frame (object) is a *kind of* the filler frame (object). Traditionally, this a-kind-of relation is specialized into two relations. The first called *Instance-Of* is used to relate a concrete object (frame) to an abstract object (frame). The second called *Is-A* is used to relate two abstract frames. We will use the convention of denoting abstract frames (objects) with capitalized names. Abstract frames correspond to *classes* in the object-oriented paradigm, and concrete frames to *instances*. There can be multiple occurrences of the above two slots, as in the following examples,

```

(hans-berliner
  ...
  <:Instance-Of ChessPlayer>
  <:Instance-Of Professor>
  <:Name "Hans">
  <:FamilyName "Berliner">
  <:Work carnegie-mellon-university>
  ...)

(Professor
  ...
  <:Is-A Academician>
  <:Is-A Employee>
  ... )

```

The *Is-A* relation defines a *taxonomy* of the concepts, in which two concepts are related by *abstraction* and *specialization* relations. The frame *Cheetah* is a specialization of *Big-Cat*, which in turn is a specialization of *Animal*, and so on. The *Cheetah* is also a specialization of *Predator*, and transitively (via *Big-Cat*) of *Animal*, and all its ancestors. This is because the frame *Cheetah* specifies more information about the set of concrete objects it represents than the frame *Big-Cat*. As a consequence, it stands for a smaller set of concrete objects (which have those extra properties) than does *Big-Cat*. In fact, the set represented by *Cheetah* is a subset of the set that *Big-Cat* stands for.

The *Is-A* hierarchy has found many applications in categorizing the kinds of things that exist, specially in the study of the natural world. The botanists and biologists of this world, of who Charles Darwin was a prominent example, have been occupied with studying life forms and categorizing them into *species*, *genus* and *family*. In our Web-enabled times, one can access such information through projects like "The Tree of Life" Web project⁸. A small sample of the vast diversity that life manifests itself in, is depicted in Figure 14.3.

The need to categorize information is of course not only confined to natural sciences, but has been the concern of philosophers and scientists in all disciplines. One must remember that the categories exist only in our heads. And so do the aggregations. They are reifications. Fundamentally, the world around us is just a vast swirl of trillions of presumably indivisible miniscule particles. Or as some somewhat extreme schools of philosophical thought believe, matter is not even fundamental. The ancient Indian schools of philosophy were concerned with existence thousands of years ago, as illustrated in the adjoining box. The Greek philosopher Aristotle espoused a method of defining general categories as the genus and the different subcategories based on the *differentiae*. In the 3rd century AD, the philosopher Porphyry drew the tree reproduced

(as shown in Figure 14.4) by Peter of Spain in 1329 (Sowa, 2006).

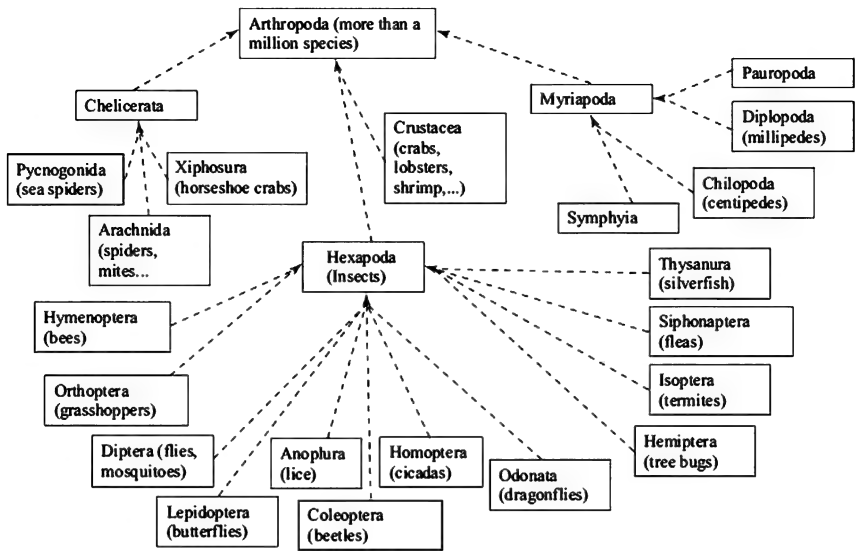


FIGURE 14.3 A small part of the tree of life categorizing small life forms, often collectively called insects. The edges represent *Is-A* relation. The leaves in the figure are in fact internal nodes in the tree of life.

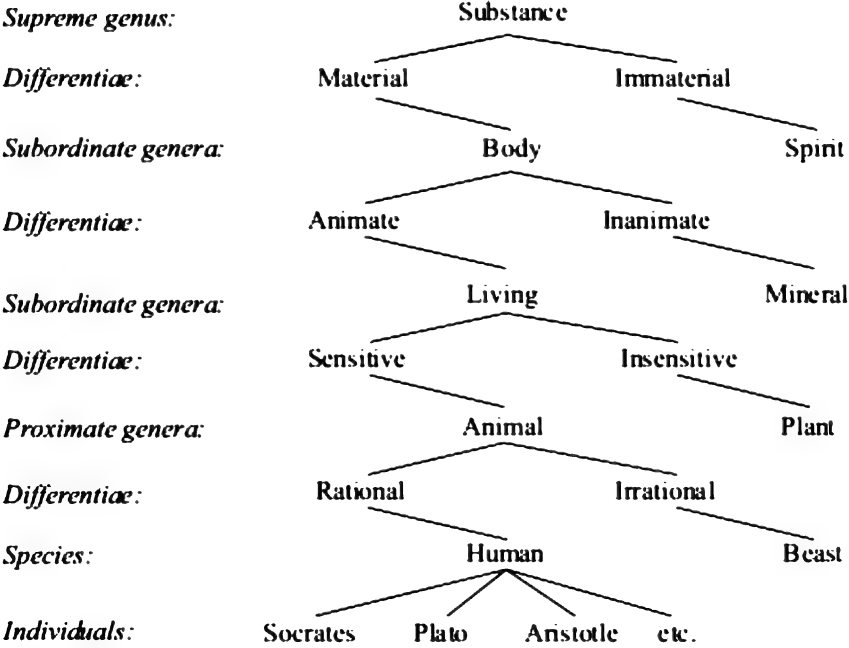


FIGURE 14.4 Tree of Porphyry, as drawn by Peter of Spain (Sowa, 2006).

Box 14.1: The Categories or Padārtha

Vaisheshika, or *Vaiśeṣika*, is one of the six Hindu schools of Vedic systems of philosophy of India (Chattopadhyaya, 1986), (Radhakrishnan, 1923). It has been closely associated with the Hindu school of logic, *Nyaya* (see Chapter 12). The *Vaisheshika* school of philosophy defines a taxonomy of all possible concepts in terms of seven individual characteristics.

The root of the taxonomy is *padārtha* (the meaning of the word), or any word that represents a concept. The concepts can be further specialized by what we call slots in the context of frames. These are seven in number—*dravya* (substance), *guna* (quality), *karma* (activity), *sāmānya* (generality), *viśeṣa* (particularity) and *samavāya* (inherence). and *abhāva* (non-existence).

These are further categorized. There are nine classes of substance—*prthvī* (earth), *ap* (water), *tejas* (fire), *vāyu* (air), *ākāśa* (sky), *kāla* (time), *dik* (space), *ātman* (self) and *manas* (mind).

And seventeen classes of qualities—*rūpa* (colour), *rasa* (taste), *gandha* (smell), *sparśa* (touch), *samkhyā* (number), *parimāna* (size), *prthaktva* (individuality), *samyoga* (conjunction), *vibhāga* (disjunction), *paratva* (priority), *aparatva* (posteriority), *buddhi* (knowledge), *sukha* (pleasure), *duhkha* (pain), *icchā* (desire), *dvesa* (aversion) and *prayatna* (effort). The following were added later, *gurutva* (heaviness), *dravatva* (fluidity), *sneha* (viscosity), *dharma* (merit), *adharma* (demerit), *śabda* (sound) and *samkāśra* (faculty).

The interested reader is referred to the (Kak, 2003) and (Narayana, 2007a; 2007b) for a deeper analysis of the representation of concepts.

14.3.1 Procedures in a Frame System

The principal advantage the frame system brings to reasoning is the savings in computation because of the explicit connectedness of related concepts (frames). Consider the inference that one may want to make that dogs are living creatures. In FOL, the relevant relations would be expressed as rules, perhaps as follows.

$$\forall x (\text{Dog}(x) \supset \text{Animal}(x))$$

$$\forall x (\text{Animal}(x) \supset \text{LivingCreature}(x))$$

Using these rules in forward or backward chaining requires the mention of an individual about who this inference is being made,

$$\text{Dog}(\text{fido25})$$

One can then go on to infer that *LivingCreature(fido25)* is also true.

Classical logic also allows us to use the Hypothetical Syllogism rule to infer,

$$\forall x (\text{Dog}(x) \supset \text{LivingCreature}(x))$$

But the real problem in making inferences is that of search. An inference engine needs to find the relevant set of matching rules and facts, to make the required inferences.

In a frame system, the task that is addressed is the same but with the aid of additional data—the pointers *linking* the different frames. From the *Dog* frame, one has only to traverse the *Is-A* links upward in search of the *LivingCreature* frame. One can imagine an algorithm that starts off at the *Dog* frame and passes a *marker* up each of the *Is-A* links. At every intermediate ancestor, more copies are made if needed, and a *mark* is left behind. The mark contains information of the immediate parent (a back pointer) and the source of the marker. The markers are tokens carrying the information that has the goal (*Dog* frame seeking *LivingCreature* frame) and keep track of the path as they traverse the links. When a marker reaches the destination frame, the algorithm can terminate. Such algorithms have been called *Marker Passing* algorithms (Hendler, 1988).

The *Marker Passing* algorithm can initiate search from both ends, sending out markers from the *Dog* frame and the *LivingCreature*. Hopefully, two markers from opposite sides will meet half way and achieve the goal quicker.

Inferences related to the part-of relation in the aggregation hierarchy can be similarly made by a marker passing approach⁹.

Another major advantage of building a taxonomy using the *Is-A* relation is that data shared between different objects can be stored with an ancestor. Here are some examples from the living world. We can store the property <Has-spine, yes> with the frame *Vertebrate*, with the understanding that this property will be shared by all its specializations. Likewise, we can store a property <Number-of-feet, 2> with the frame *Biped*, the property <Maintain-body-temperature, Yes> with the frame *Homeothermic*, and the property <Lays-eggs, Yes> with the frame *Oviparous*.

A frame stores a property value <slot filler> pair that is common to all its descendants. As a consequence, the descendent frames inherit the property and value from the frame. In Figure 14.5 below, the examples given above are shown along with some descendants. *Homo-sapiens* are descendants of *Homeothermic*, *Biped*, *Mammals* and *Vertebrates*. Consequently, human beings (*homo sapiens*) have the property of being warm-blooded (*homeothermic*) mammals with vertebras and two legs, apart from the explicitly stated property. *Birds* share three of the ancestors (in the taxonomy) of *Homo-sapiens* and have in addition the property that they lay eggs.

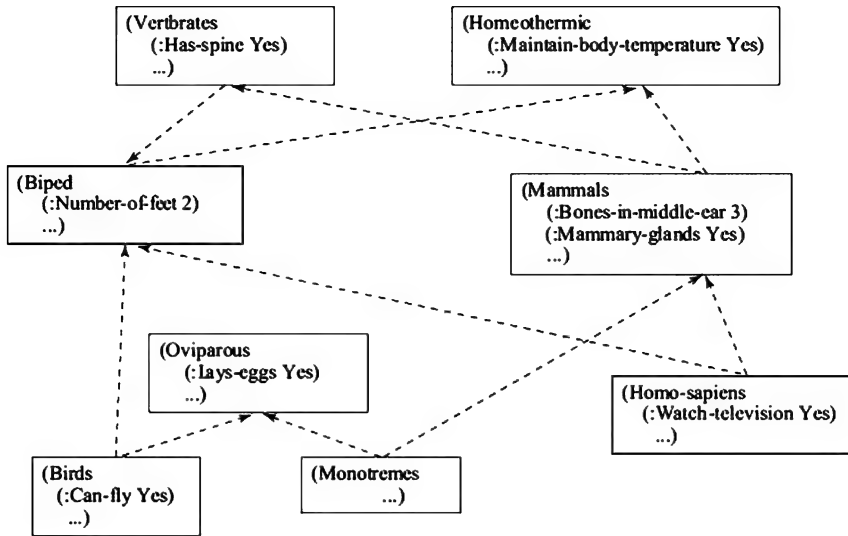


FIGURE 14.5 A part of the taxonomy of life. A property stored in a frame is inherited by all its descendants. The Is-A relation is represented implicitly here by the dashed arrows.

A frame system should be implemented with a generic procedure for inheritance that will propagate property values to all the descendants. This can be done in a data-driven manner.

- Whenever a property value is added to a frame, or modified, the procedure should traverse the subtree containing its descendants and add the property value to each descendant frame.
- When a new frame is inserted in the frame system, the procedure should traverse the subtree of all its ancestors, and copy property values from each ancestor.

This can also be done in a demand (goal) driven manner. Whenever the fillers of a particular frame are needed, a traversal of all its ancestors will yield the fillers to be inherited.

Inheritance of slot fillers is not mandatory. A child frame can override the value it receives from an ancestor and specify its own filler. For example, the frames *Penguins* and *Ostriches* would be children or descendants of the frame *Birds*, but will not inherit the filler “Yes” for the slot “Can-fly”. In this manner, we can assert that birds can fly to serve as a *default* statement, but we can add exceptions to birds like penguins and ostriches. This allows us to make general statements like “birds can fly”, “leaves are green”, and “mushrooms are edible”, without having to commit them to be universal statements.

The frame system is a mechanism for organizing knowledge. A consequence of allowing arbitrary slot filler values for frames is that the onus of correctness now lies entirely with the user. The frame system simply serves as a means for economy of expression. Along with economy also comes consistency. If a property value were to be modified, then it needs to be done only in one place. Property values are

not usually modified when representing knowledge about the (unchanging) world, for example the living world. In such a world, we can take recourse to a stricter representation mechanism that does not allow a user to make inconsistent statements, but forces her to be logically consistent. For example, we can associate “*Can-fly*” property with a class called “*Flying-creatures*” and make “*Flying-Birds*”, and “*Bats*”, its children. Then “*Flying-birds*” can be children of “*Birds*”, but they inherit the flying property not from “*Birds*” but from “*Flying-creatures*”. We will explore such *Description Logics* later in this chapter.

Meanwhile, frame systems can be used to represent information rich in abstraction and aggregation. In addition to relations giving rise to the two hierarchies, one can have other relations between frames that are constituents of a larger frame. Frame systems can also be used to compose complex event patterns.

Consider the task of organizing a birthday party for a child, and the knowledge needed for doing so. The entire activity can be organized around a *Birthday-Party* frame that captures a typical birthday party (see Figure 14.6). The frame says that a *Birthday-party* is a *Party*, and by inheritance has food and costs money. In addition, the *Birthday-party* must have a *B'day-kid* for whom the party is organized and who is a *Person*. The frame has a *Date* which is the birth date of the *B'day-kid*, which can be obtained from the instance when creating an instance of the *Birthday-Party*. A *Birthday-Party* must have a slot called *Guests* whose filler is a set (or list) of *Guests*. *Guests* are also *Persons*, and they must be *Friends* of *B'day-kid*. The name of the guests can be obtained from the names of friends of the instance of *B'day-kid*. Finally, a *Birthday-Party* has activities that happens before, during, and after the party, represented by the three corresponding slots. The first of the three *Preparation*, must begin 10 days before the birthday. It has tasks like choosing the guest list, sending out invites, ordering the food and the cake, and choosing games and music. The *Preparation* frame is followed by the *PartyEvents* frame that describes the happenings during the party, starting with receiving guests and ending with sending them off with return gifts. All this is followed by *After-the-party* frame that describes the cleaning up activity and the excitement of opening the presents. There may be activity the day after as well, for example sending thank you notes.¹⁰

Figure 14.6 below shows a part of the Birthday party frame system. The descriptions of the games involved and the time needed for them, the food menus, the cake types, and decoration details have been left out for want of space in the figure.

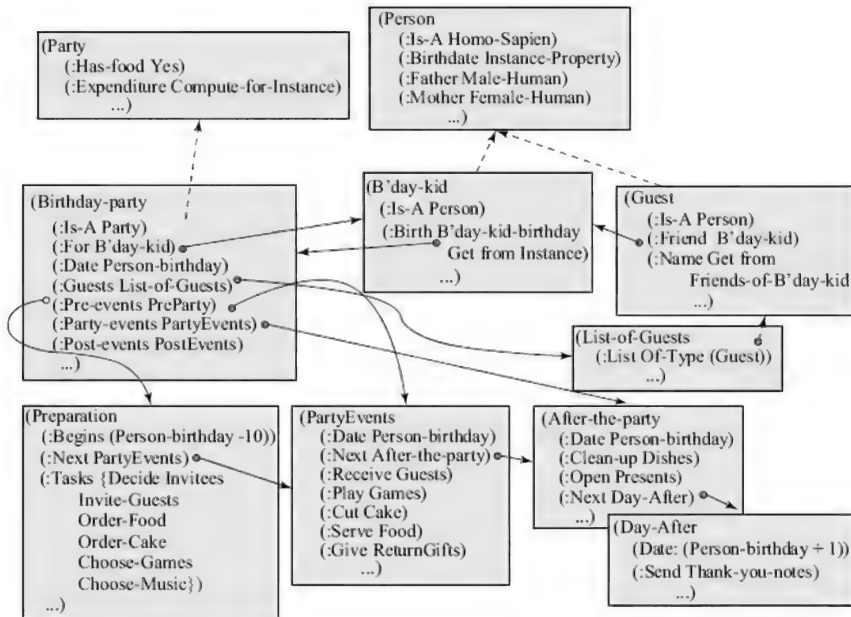


FIGURE 14.6 A part of the birthday party frame system.

Such a frame-based representation could be augmented with procedures that carry information from one frame to another when it gets generated. Information can be conveyed in both the *eager* mode and in the *lazy* mode. In the eager or data-driven mode, a procedure with an *If-Added* test is triggered when a filler is added to a slot or modified. For example, when an instance of a birthday party is being created, as soon the kid's birthday is entered, the date could be propagated to various frames that have a slot for that date. The lazy or goal-driven could have procedures invoked by an *If-Needed* trigger. Such a procedure could pull in data from different frames. For example, if one needs to compute how much expenses will be incurred then a procedure could traverse frames containing expense related information and sum it up.

Such a frame system could then serve as a kind of an *active hierarchical spreadsheet-cum-calendar* that could help one plan a party meticulously, and generate calendar reminders when needed. When invoked, the procedures would facilitate the construction of actual instances of the frames for a specific birthday party. The process could happen somewhat along the lines described below. We assume that instance frames of the people involved, food items, and so on exist in some database.

Consider the task of organizing a party for Aditi on November 7. The moment we invoke the birthday-party organizer, it creates an instance of the Birthday-party frame, as follows,

```
(birthday-party-4327
  (:Instance-Of Birthday-party)
  ...)
```

and prompts us for the name of the child, by showing us a list of names in our database, or asking us to enter the data. It retrieves, or constructs, the instance frame for Aditi, and creates an instance of the *B'day-kid* frame,

```
(b'day-kid-aditi
  (:Instance-Of B'day-kid)
  (:Birthday november-7)
  ...)
```

The system would then create instances of the three event frames. For the instance of the *Preparation* frame, it would initiate the task of forming the guest list. Assuming that we have an existing database of people we know, the system can show us a list of friends of Aditi, and ask us to choose the guest list. As each guest is chosen, the system creates an instance of a *Guest* frame for example as follows,

```
(guest-subun
  (:Instance-Of Guest)
  (:Friend aditi)
  (:Name subun)
  ...)
```

It would create a calendar entry for inviting each guest, which would prompt the user at an appropriate time. A *smart* system may also retrieve food preferences of each guest if available and impose soft constraints on the party menu. It would then show you menu options, perhaps on the Web, and help you decide the food items, calculating quantities, costs, and if you insist, calories. It would show you the games menu and likewise help you decide the entertainment program, initiating calendar entries for buying anything that might be needed, and adding up the estimated time for each event. It would finally help you choose and order a cake. The top-level frame could pull in the requisite data from the constituent frames, via an *If-Needed* trigger, and display the total estimated expenses and duration of the party. The system would then create a complete frame instance, which would prompt you during and after the party on your mobile phone. Figure 14.7 shows a part of the final, concrete frame system created.

The key thing is that a frame holds together all the information related to structured event patterns together. Thus, when one thinks of such an event, all aspects related to it are brought to the fore. Computationally, this will be efficient when the links are implemented using a data structure so that they can be traversed directly.

Cognitively, the frame acts like a schema, chunking together all related information. If someone were to announce that she has just been to a birthday party, then a curious aunt might ask her all sorts of questions—Whose birthday was it? Who all were invited? What gift did you take? What games did you play? How was the food?—and so on. Such questions come readily to mind because one already has a preconceived idea of the entire gamut of activity that constitutes a birthday party.

A number of research students working with Roger Schank at Yale used this notion of chunking together of data related to stereotypical event patterns to write programs to understand stories in natural language. The key exercise was to create representations of such *knowledge structures*, called *Scripts*. Retrieving and using an appropriate Script makes the task of story understanding a feasible one. We look at Scripts and other related knowledge structures after the following section.

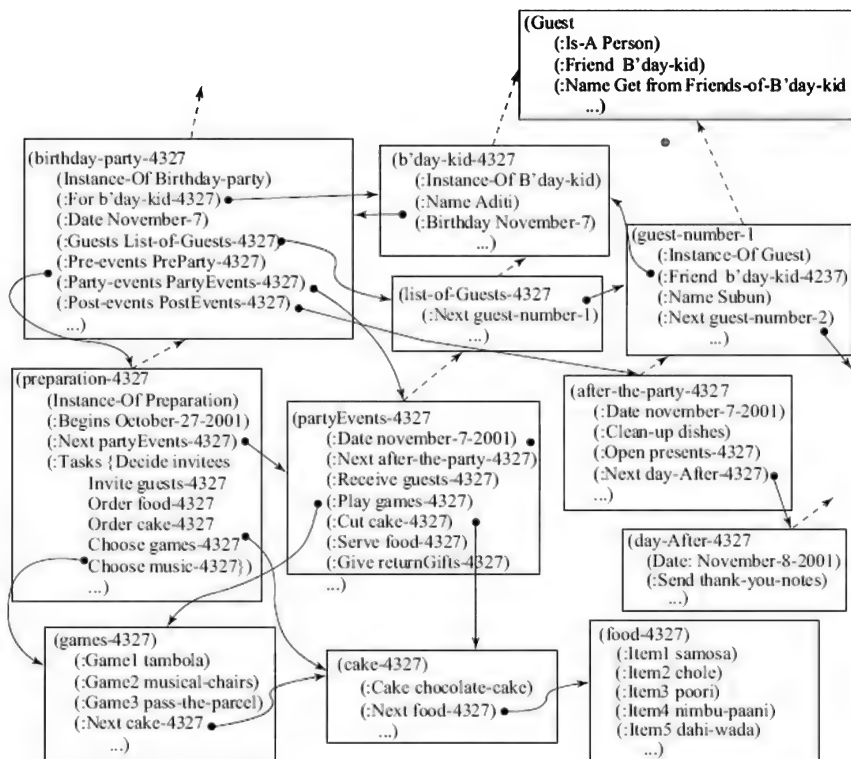


FIGURE 14.7 A part of the concrete-frame system for a specific birthday party.

14.4 The Semantic Net

The aggregation hierarchy only captures the *part-of* relation, and the

Scott Fahlman (1980), (Fahlman et al., 1983) explored the idea of hardwired implementation of the activation spreading networks with the system NETL. The idea of a massively parallel activation spreading network has inherent appeal because our own brains can be viewed as such networks. Fahlman also introduced the distinction between individuals and classes, the notions of inheritance, and also the idea of overriding the inherited values. The oft-quoted example of Clyde being an elephant and therefore being grey can be traced to papers by Fahlman. The concepts of activation spreading were further explored by many researchers (see for example (Collins and Loftus, 1975), (Anderson, 1983), (Hendler, 1988), (Wolverton and Hayes-Roth, 1994), and (Jiang and Tan, 2006)).

Eugene Charniak carried forward the idea of marker passing to processing of language. One of the key problems in natural language processing, known as the problem of word-sense disambiguation, is to choose the sense of the words in a sentence. The problem arises if one is trying to write a language comprehension program. In an earlier paper, Charniak (1977) describes a frame-based story-understanding program he called *Ms. Malaprop*. The paper discusses the vast amount of knowledge about painting that a system will need to have to be able to comprehend stories about painting. For example, the system should “know” that before one paints an object, one normally cleans it to avoid flaking, or that if there is too much paint on the brush, it might drip. The painting frame discussed by Charniak knits together all such information. The process of understanding a story, or comprehending a sequence of utterances in a language is “*fitting what one is told into the framework established by what one already knows*” (Charniak, 1977).¹²

We will look at this process in a little bit more detail when we look at Schank’s work below. The level of understanding that a system (or a human for that matter) has, depends upon the depth of knowledge the system (or human) has. Having knowledge of physical objects like the facts that paint is liquid and can drip, and that paints dry up, help in understanding why things happen the way they do. A deep understanding system would have the typical event patterns captured in frames, with specific events linked to domain knowledge like physics or chemistry, to help explain why things happen the way they do.

The idea of using words of a language to construct networks of relations was further taken up in the *Wordnet* project at Princeton (Fellbaum, 1998) described in Chapter 16.

The traditional approach to language processing was to first do a syntactic analysis of the given sentence to determine the grammatical structure, and then the semantics. Charniak (1983) proposed that the networks for semantics and syntax of words and phrases from a language be interconnected to enable information flow from syntax to semantics and vice versa (Figure 14.9). Word-sense disambiguation could be done by passing markers from each of the senses of the word that are possible. The markers from the correct sense of the word were

more likely to intersect with those from the other related words. This idea for *word-sense disambiguation* was also developed by Graeme Hirst (Hirst and Charniak, 1982), (Hirst, 1984; 1988).

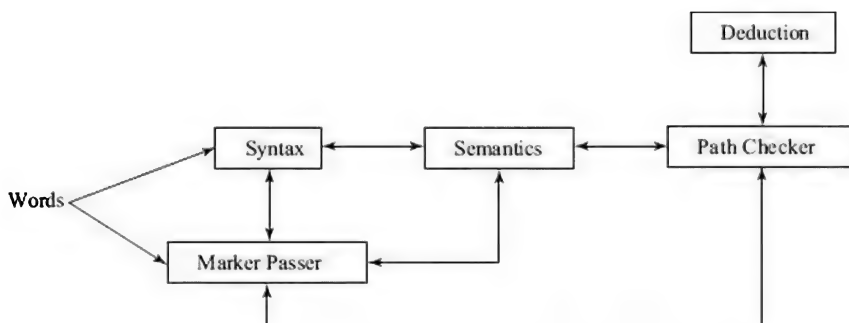


FIGURE 14.9 The interplay between different kinds of information and processes in Charniak's proposed parser (Charniak, 1983).

The marker passing algorithm is essentially a form of search with distributed *control*. A related idea is that of *relaxation*, which comes from connectionist systems like the Hopfield network (see Chapter 4). In such a network, nodes can be in different levels of activation, and the activation value of a node influences the activation value of its neighbours via the edges. The influence can be *excitatory* or *inhibitory*. Observe that the edge now represents whether the two nodes at its ends support each other or oppose each other. Given an initial excitation of some nodes, the system is allowed to relax by propagating influences across edges, till it settles down into a steady state. The set of excited nodes can then be seen to describe the state of affairs.

The network in Figure 14.10 is the kind of network proposed by David Waltz and Jordan Pollack to combine the syntactic and semantic relations associated with a string of words (Waltz and Pollack, 1985). The figure shows a part of the network that would be associated with the celebrated sentence "Time flies like an arrow" (Kuno and Oettinger, 1962). The words in the boxes represent the input sentence that one needs to understand, and the different arcs link the words to syntactic or semantic categories. As one can see, the sentence is ambiguous. The word "time" for example, could be an adjective, a verb, or a noun. When "time" is a verb or an adjective, "flies" is a noun, but when "time" is a noun "flies" is a verb. These different possible relations are expressed by the excitatory and inhibitory links between the nodes.

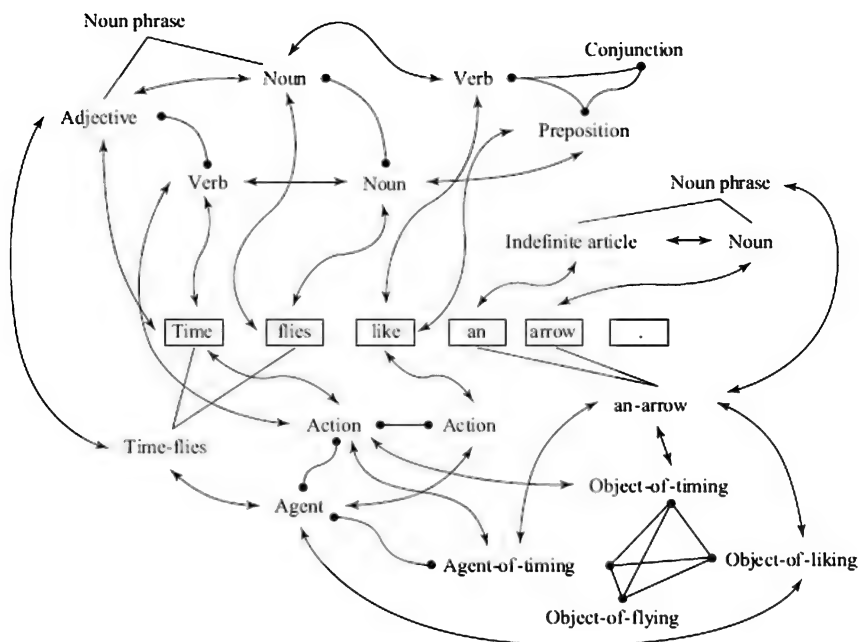


FIGURE 14.10 Combining syntax and semantics into a connectionist network. A part of the network for the sentence “Time flies like an arrow.” is shown above. The nodes above the words are part of syntax and the nodes below define the semantics. Edges with arrowheads represent excitatory links and edges with rounded ends represent inhibitory links.

To get a particular interpretation, one might activate certain nodes. For example, if “Time” and “adjective” are activated then the verb sense of “time” will get inhibited via the link, the noun sense of “flies” will get activated, which will inhibit the verb sense of “flies” (not shown in the figure). The process of relaxation is to allow all nodes to be influenced by their (edge) neighbours, till they settle down into a stable state. In that case, one expects one sense of each word to get chosen in a consistent manner, along with the appropriate syntactic categories.

Amongst the ideas put forward by Quillian was that instead of using different kinds of edges to relate concepts, the relations (usually verbs) could themselves be concepts, and represented as nodes in the semantic net. This notion was further developed by John Sowa in his work on *Conceptual Graphs*. We look at *Conceptual Graphs* in the last section in this chapter.

14.5 Scripts, Goals, Plans and MOPs

The ability to use language has been hypothesized to have been critical to the development of intelligence in humans. It is likely that this faculty will need to be bestowed upon machines too, if they are to ever operate intelligently in a dynamic multi-agent world.

It has been realized by many researchers in artificial intelligence that a crucial, and probably critical, part of language understanding is the ability to represent knowledge and reason with it. Language is in fact a means of exchanging and storing knowledge, but it is the knowledge-processing ability that is the basis for the use of language. And a key to using knowledge efficiently is using structured representations. This has been the theme of this chapter beginning with schemas and frames, and this theme has been brilliantly illustrated by the work of Roger Schank and his group at Yale University that we now look at.

Human discourse is surprisingly economical. Very young children need things to be spelt out in detail, but as they acquire more and more knowledge, one needs to say less and less.¹³ Communication between adults involves a considerable amount of reading between the lines. And this is possible because of the knowledge structures shared between the interlocutors.

Consider the following two sentences.

Arvalan walked into the store. He picked up a jar of pickles and went home.

A linguistic analysis by itself would not yield much because the two sentences say little explicitly. That someone (or something) called Arvalan walked into “the store”. And then a sentence saying that he picked up “a jar of pickles” and went home. Semantics, or the meaning of words, helps us understand the sentences individually (see Chapter 13). It is only world knowledge or pragmatics that is instrumental in us being able to understand the whole story. Our knowledge of what is a store and what (typically) happens in a store helps us understand that Arvalan is (probably) buying a jar of pickles, and the store is a place that sells jars of pickles, amongst other things. Our knowledge of what pickles are gives us cues of what Arvalan might do with the pickles. If these sentences were accompanied by others like “He was sick of the hostel food” or “He was planning to visit his friend in Mumbai”, we might be able to make an educated guess of what his intentions were.¹⁴

Observe that any such inferences we might make are only probable inferences, signifying what *typically* happens in a store or what are pickles used for. They define what one would *normally* expect the sentences to pertain to. It is possible however that the inferences do not correspond to what is really happening. For example, Arvalan could have been a detective investigating a crime, or a robot being tested in the real world. But the most likely scenario is that Arvalan is a shopper going in to buy a jar of pickles. And it makes economical sense to quickly being able to arrive at such inferences towards understanding the sentences.

Roger Schank and Robert Abelson proposed that knowledge of *stereotypical patterns of activity* is packaged into structures called *Scripts* (Schank and Abelson, 1977), (Schank and Riesbeck, 1981). A Script is a structure that captures the pattern of activity, and like a theatre or a movie script, has *roles* and *props*. The Script lists out all the actions that

are *expected* to happen in the situation. Thus, when a Script is invoked, it generates *expectations* (see also Chapter 13) of what events or actions will happen. For example, in a shopping Script, the main roles are of the customer and the shopkeeper, and the props include the shop, the counter with the cash register, and the merchandise. The events that one expects include the customer asking for something, or surveying shelves for some things, picking objects and possibly loading them in a basket, and checking out at the counter by paying the requisite amount of money via a card or in cash. Figure 14.11 illustrates the structure of a shopping Script. The Script is made up of a sequence of episodes with possible branches in the flow of events. In the figure, each episode is described in English and with a semiformal statement of the main conceptualization in the episode (drawn in the style in (Schank and Riesbeck, 1981)). The conceptualizations are expressed in *Conceptual Dependency*. The variables with an "&" prefix are either roles or props.

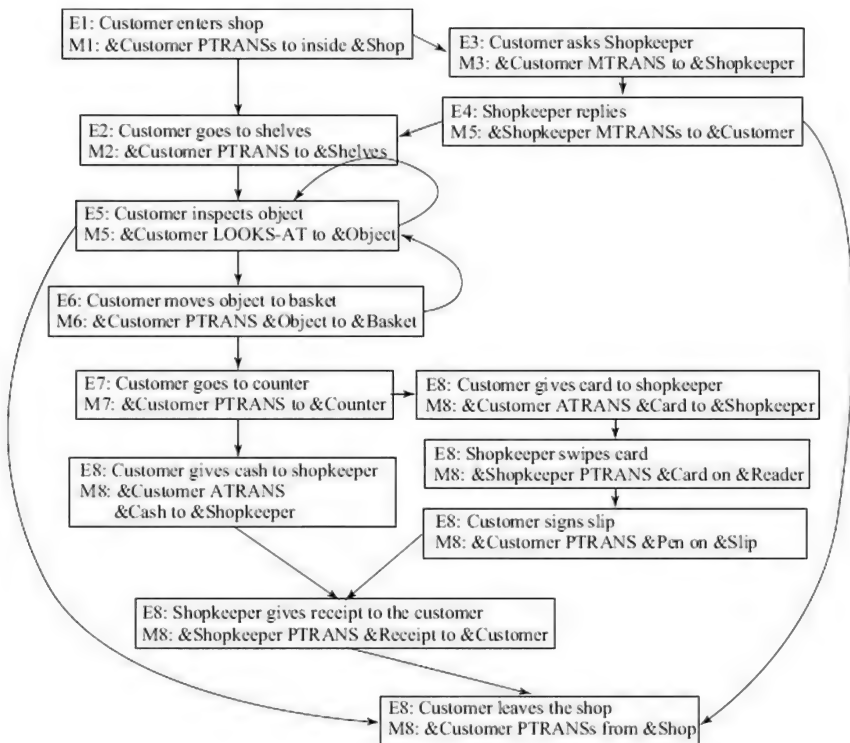


FIGURE 14.11 A shopping Script. Each episode written in English has a main conceptualization expressed in the Conceptual dependency theory.

Like Frames, Scripts are a mechanism for chunking and generating expectations while listening to language utterances. The moment one hears of someone wanting to buy something, or someone going into a shop, or someone having been shopping, one can retrieve the shopping

Script. This will now serve as the structure into which subsequent utterances can be expected to match.

Richard Cullingford (1977; 1981) wrote the program called *Script Applier Mechanism* (SAM) that implemented Script-based story understanding. The program received input from the *English Language Interpreter* (ELI) (Riesbeck, 1975) (also see Chapter 13) and gave its output to a program called BABEL an English generator (Goldman, 1975) that took CD conceptualizations as input. SAM itself was concerned only with Script application as illustrated in Figure 14.12, where the conceptualizations generated are depicted schematically. Given that a Script has been retrieved from memory and instantiated, the conceptualizations generated from the input sentences should match the corresponding conceptualizations in the Script. If that happens then the Script is applicable and generates the scaffolding for the overall story.

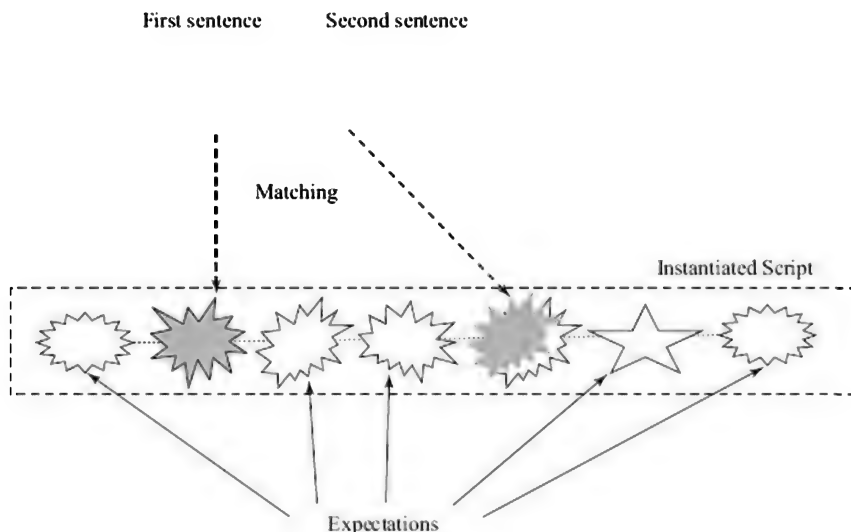


FIGURE 14.12 Matching Scripts. Conceptualizations generated from input sentences should match conceptualizations in the Script. The remaining conceptualizations in the Script serve as expectations on what one might hear.

SAM needs to constantly verify that the Script it is using is indeed the Script that explains the story. It does this by maintaining a score that is an indicator of its belief in the Script. SAM also keeps other Scripts that could be applicable in its short-term memory. SAM fetches Scripts from its repository by a process of matching cues. Cullingford (1981) describes the following patterns used by SAM to retrieve scripts, with illustrations from a subway travel script. These patterns are stored as headers in the script.

Direct Header Mentions the explicit use of the Script. For example, “*John took the subway to Coney island*”.

Locale Header A circumstantial cue, for example, “*John walked into the Boro Hall subway station.*”

Instrumental Header When the Script usage is mentioned as an instrumental action. For example, “*The IRT took John to Shea Stadium.*”

Precondition Header Here, a sentence indicates that the preconditions for applying the Script hold. For example, “*John wanted to go downtown.*”

Scripts contain conceptualizations that one assumes implicitly to happen, and may serve as receptors if more details in the story are forthcoming. They may also serve as plausible inferences about what might have happened, even though it was not stated explicitly. The following example from Cullingford’s work shows how SAM could answer questions in a knowledgeable fashion.

Input story (Cullingford, 1981)

“John went to a restaurant. He ordered a hot dog. The waiter said they didn’t have any. He asked for a hamburger. When the hamburger came, it was burnt. He left the restaurant.”

The following is the question-answer session with SAM (assisted with English by ELI and BABEL).

Q1: Did John sit down in the restaurant?

SAM : *Probably.*

Q2: Did John order a hot dog?

SAM: *Yes.*

Q3 : Did John eat a hot dog?

SAM: No, the waiter told John the management was unable to give it to him.

Q4 : What did the waiter serve John?

SAM: The waiter served John a hamburger.

Q5 : Why didn’t John eat the hamburger?

SAM: Because the hamburger was overdone.

Q6 : Did John pay the check?

SAM: No. John was angry because the hamburger was overdone and so he left the restaurant.

Observe that the Script helps interpret sentences like “when the hamburger came” correctly as the hamburger being in fact served by the waiter. This is partly because SAM has access to a semantic module that can match categories with actions, and partly because the role definitions in the script. Both dictate that an inanimate object cannot move by itself and has to be carried by an animate one.

Answering questions is not a straightforward task. First, one has to

understand the question and figure out what the asker wants to know, and then supply the pertinent information. For instance the correct response to the question “Could you tell me the time?” is the time at that moment, and not “Yes”, even though the latter seems on surface to be appropriate. Many such issues of question answering were investigated by Wendy Lehnert in her doctoral thesis as part of the Yale group (Lehnert, 1978).

14.5.1 MOPs

A Script is a data structure that stands in isolation. As the problem solver’s experience increases, the number of scripts in its memory will increase. Memory Organization Packets (MOPs) were introduced to weave the many possible scripts into one large structure (Schank, 1982; 1999). The idea, like in Frames, is not only economy of representation, but also to facilitate the finding of relevant knowledge.

Consider a linear script like a visit to a dentist. The main action takes place when the patient goes and sits on the dental chair; the dentist inspects her teeth, and carries out the required procedure. This activity may be grouped together in a *scene*. Schank defines a scene as “a memory structure that groups together actions with a shared goal, that occurred at the same time”. Further, “a MOP consists of a set of scenes, directed towards the achievement of goal. An MOP always has one major scene whose goal is the essence or purpose of the events organized by the MOP” (Schank, 1982). A dentist MOP then would consist of four scenes: making an appointment, arrival and waiting, consultation, and payment. A visit to a lawyer MOP, shown in Figure 14.13, would share three scenes with the dentist MOP and differ on in the consultation scene. While the dentist scene may have a tooth-extraction event, the lawyer scene may have a contract-signing event. The memory structures are themselves organized into *packaging* (aggregation) hierarchies and *abstraction* hierarchies, as shown in Figure 14.13.

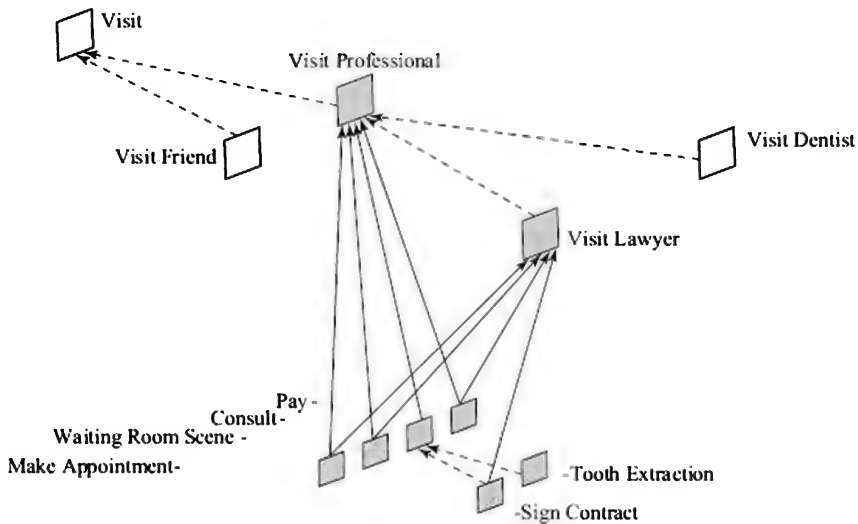


FIGURE 14.13 Memory organization packets are made up of abstraction and packaging (or aggregation) hierarchies.

At this point, we can distinguish between two kinds of knowledge that exists in an agent's memory. One, *semantic memory*, is ontological in nature, and defines the categories and instances of things and processes that exist and the relations between them. The other is *episodic memory* that stores the experiences of the agent. This is the kind of memory a case based reasoning agent works with (see also Chapter 15). Episodic memory is the memory of episodes. Episodes are made up of events. And as discussed in Chapter 13, events can be thought of as belonging to another class of things. Another distinction between episodic and semantic memory is that the former is dynamic while the latter is more or less static. However, this distinction is blurred in the MOP. This is because the MOP can store both classes and instances at different level of granularity. Episodes (that occur) are sequence of instances. They can find a place in the MOP in which each event, or sequence of events, can be an instance of a scene or a script. In fact, it has been hypothesized that structures like scripts are learnt via repeated occurrences of similar experiences, and this can be seen as a process of learning abstract knowledge from instances.

Not surprisingly, a lot of work on MOPs was reported in the context of case based reasoning (Riesbeck and Schank, 1989), (Kolodner, 1993). The fact that a MOP is an integrated structure, containing all concepts and actual experiences allows for the possibility of cross contextual reminding. This can happen when episodes in different domains or contexts have a common abstraction level at which their similarity can be observed. This was demonstrated in a system called SWALE (Kass, 1986), (Leake and Owens, 1986) that explained the death of a successful and healthy race horse called Swale by matching the episodes with

another story (case) where a “spouse kills spouse for life insurance”, substituting the spouses with the owner and the horse. Another program that deployed a MOP-based memory was the program called CYRUS written by Janet Kolodner (1984) that kept track of the travels of the diplomat Cyrus Vance. Faced with the query “*Did you ever meet Mrs. Begin?*” Cyrus generates a question “*When would I meet the spouse of a diplomat?*” and searches into its memory for instances of having gone for diplomatic visits to Israel. Another program that exploited the integrated nature of the MOP was DMAP (Direct Memory Access Parsing) by Charles Martin (Riesbeck and Martin, 1985), (Martin, 1989), which worked on the assumption that the principal task in parsing language was finding the relevant memory structures. In fact, Martin says that the “output of a case based parser is a new state of memory”. Figure 14.14 below, adapted from (Lehnert, 1987), shows a part of the memory that is activated when the sentence “Milton Friedman says interest rates will rise.”

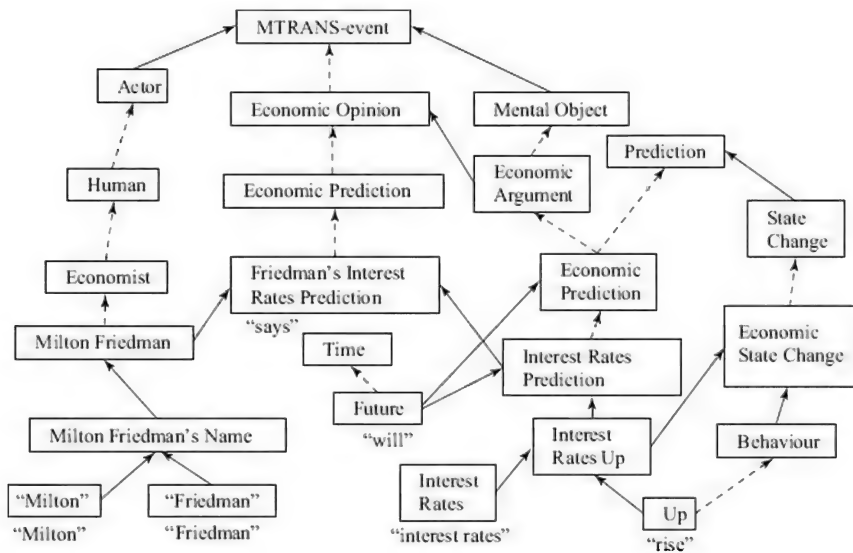


FIGURE 14.14 An illustration of MOP activation by DMAP (figure adapted from (Lehnert,1987)).

DMAP and also the parser by Waltz and Pollack mentioned earlier, views the memory as a vast integrated network incorporating different kinds of knowledge, and views parsing as a process of activation, and change, of this memory. Every sentence that the listener hears leaves the memory in a changed state.

14.5.2 Goals, Plans and Actions

Scripts are patterns of stereotypical activity. They help a listener—whether human or machine—tie up sequences of sentences together into

a coherent story. We can understand the first two sentences in the restaurant story because ordering food is part of the restaurant script. The actors in a story matching a script (are expected to) do what the script says they do. Understanding of sequences of sentences is successful matching with the pattern that is the script. The connections between the sentences are defined by the script. However, script based understanding does not give us any insights into *why* the actors are doing what they do. For that, we need to represent connections that track the intentions that lead to the actions of actors. Consider the following sequences of sentences.

A. Akimi was feeling hungry. She reached out for the phone.

B. Anandan wanted to get rich quick. He decided to join politics.

Now it would be difficult to imagine scripts catering to these sentences. There might well be a stereotyped sequence of activities that results in satisfying hunger, but there might be too many of them¹⁵, and the argument is the same for getting rich quickly. On the other hand, we should recognize that some sentences talk about goals, some describe plans, while others describe actions. We need to establish connections between the goals and the plans and the actions. For example, Akimi might be about to order some fast food, or ask a friend for going out, or even call up home to announce her impending arrival. Without being able to make such connections, there is no way one could have connected the two sentences. What we need is knowledge of *how* goals, plans and actions are related. This is what the theory of goals and plans expounds (Schank and Abelson, 1977), (Wilensky, 1981; 1983).

People rarely act randomly. Usually their actions are part of some plan. If you know what the plan is then you can *understand* the action. For example, if one is executing a plan for getting food from a restaurant, a constituent action may be to call them to order food. Then reaching out for the phone can be understood as an action that is part of that plan. Or if you perceive the school bully walking towards you in a menacing manner, you can infer that he has a plan of snatching your new toy truck. Thus, actions can be fitted into a story, if they are part of known plans. Plans, on the other hand, are usually devised to achieve some goals. If a child has a plan for reaching up to the cookie jar, it must be because he has a goal of getting hold of a cookie. Goals, like getting hold of a cookie, themselves may be part of higher level plans, which in turn may be part of higher level goals. The actions of the child can be “explained” by the fact that the goal of getting hold of a cookie is part of the plan of eating it, to achieve the goal of eating a cookie, which is a part of a plan to do something to satisfy hunger, which is to achieve the goal of satisfying hunger. Figure 14.15 illustrates this idea with two other sentences.

1. Balaji needed money to buy a music system.
2. He called his sister.

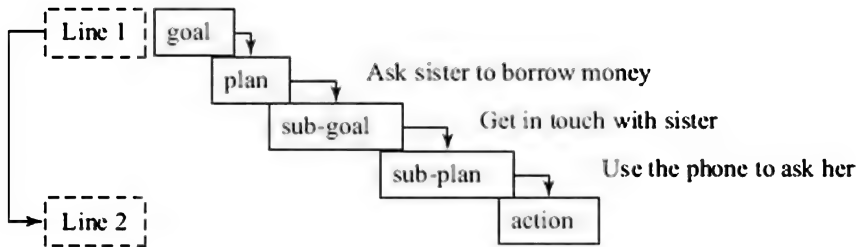


FIGURE 14.15 The goal plan relations sitting behind an action. Given lines 1 and 2, an astute listener might make the connections as shown.

The task of understanding stories can then be seen as finding the connections between sentences. Sometimes, as in the above figure for example, the story begins with a stated goal. But often it may not. To make sense of such stories, one must keep a set of permanent goals that people always have. These are goals like happiness, wealth, possessions, maintenance of health, and recurring goals like satisfying hunger.

A program called Plan Applier Mechanism (PAM) was written by Robert Wilensky for his doctoral thesis at Yale (Wilensky, 1981; 1983). The knowledge of relations between plans, goals and actions was represented in the form of rules, called *requests*. The rules have the following form.

Action → Plan
 Plan → Goal
 Goal → Plan

The rules or requests are explanatory or *abductive* in nature. The reading of the rule *AEF* is “if one sees or hears about the action *A*, then maybe *A* is happening as a part of the plan *P*≤. And likewise, for rules relating plans to goals, and goals to plans.

Each *active* request of the form “pattern → action” specifies two things¹⁶. One, the “pattern” that signifies PAM’s expectation in terms of the conceptualization PAM expects to see. At any given moment, PAM may have more than one *active* request waiting for input. Each request has an expectation attached. The other thing that the request specifies is the “action”, which is the processing to be done when the expectation is met, that is, the expected conceptualization appears in the input. The action may be the creation of new requests, filling up of slots in some structure, adding to the story representation or deleting other requests. Slots are called *gaps* in the Yale group literature. There is a special gap called the *Input* gap that serves as the immediate memory for PAM.

Processing of a natural language sentence begins by ELI (Riesbeck, 1975) parsing the input sentence and placing the resulting conceptualization in the *Input* gap. PAM then works in three modes.

1. Predictive Mode If the conceptualization in the *Input* gap matches one of the active expectations then it accepts it, executes the actions specified with the request, and goes into the Incorporation mode. PAM was expecting the conceptualization to appear, and when it does, it adds it to its explicit account of the story.

2. Bottom-up Mode If the conceptualization placed in the *Input* gap does not match any expectation, PAM strives to produce one that does. It does this by looking for a request in its repository whose pattern matches the input¹⁷. If no request exists, PAM reports failure. If there is one then the action of the retrieved request points to another conceptualization that is placed in the *Input* gap. PAM then goes back to the Predictive mode.

3. Incorporation Mode PAM enters this mode, after it has found a conceptualization in its *Input* gap that it was expecting. The conceptualization could have been produced directly from the input sentence, or it could have been produced by a chain of request firings in the bottom-up mode. Starting from the input conceptualization to the one that matched an expectation, the entire chain is appended to the story being recreated, and more requests may be activated in the process.

The working of PAM is shown schematically in Figure 14.16, when the input conceptualization describes an action.

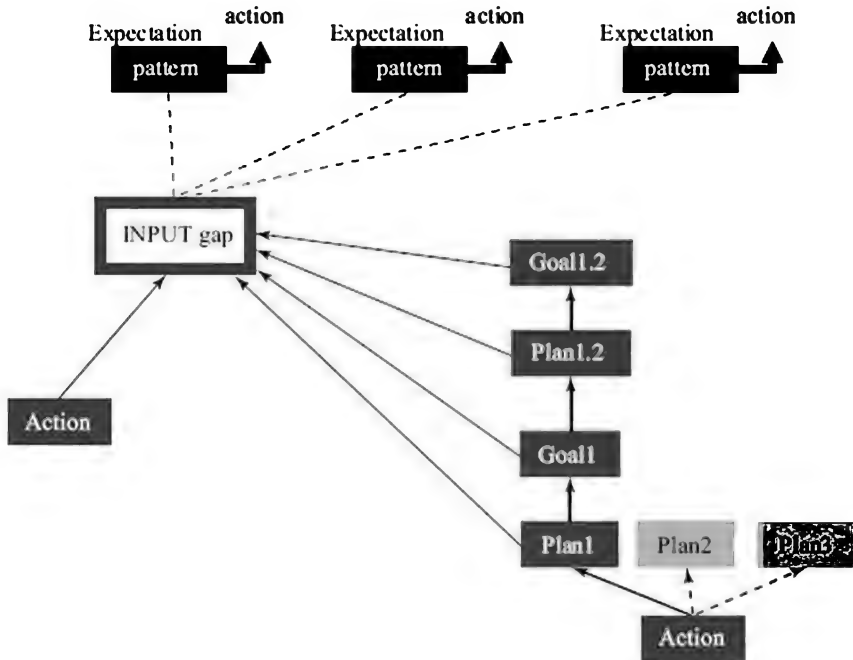


FIGURE 14.16 PAM has a set of expectations that are patterns associated with active requests. Either the input conceptualization or something derived from it in the bottom-up mode is placed in the Input Gap for PAM to test in the Predictive mode. If it matches an expectation, PAM goes into the Incorporation mode.

We look at a small example described in (Wilensky, 1981) of how PAM processes sentences. The two sentences are,

John was hungry. He ate at a restaurant.

ELI produces the following conceptualization for the first sentence,

((ACTOR HUM0 IS (*HUNGER* VAL (-3))))

This is a state description, and does not meet any expectation. PAM goes into the Bottom-up mode and searches for a request that has the “hungry” state as a *pattern* or *condition*. It finds the following,

HUNGER-RULE

```
Condition: ((ACTOR HUM0 IS (*HUNGER* VAL (-3))))
Action:    ADD GOAL:  S-HUNGER PLANNER ( )
           SOURCE: THEME = HUNGER-DRIVE
           PLAN: ( )
Suggestions: Set TARGET = PLAN
            To FOCUS = !INPUT!
            Using SUITABLE-PLAN-RULE

            Set TARGET = PLANNER of GOAL
            To FOCUS = ACTOR
            Using FOCUS-REQ
```

PAM applies the rule and places the following conceptualization in the *Input gap*,

```
(S-HUNGER    PLANNER ( )
          SOURCE THEME = HUNGER-DRIVE
          PLAN ( ) )
```

The source of the goal S-HUNGER is the *theme* HUNGER-DRIVE. A theme is something that PAM accepts as something that happens in a recurring manner. It does not need any further explaining. People have needs that need to be periodically satisfied, and hunger is one of them. The "S" prefix signifies that it is a satisfaction goal. For goals that derive from themes, PAM seeks no further explanation. It goes into the incorporation mode, and adds the S-HUNGER goal to the story. Observe that there are two empty *slots* or *gaps* in the added structure, the ones named PLANNER and PLAN. PAM activates *requests* called *suggestions* to fill these slots.

The first suggestion in the HUNGER-RULE is,

```
Set TARGET = PLAN
To FOCUS = !INPUT!
Using SUITABLE-PLAN-RULE
```

This says that the gap or target of this request is the PLAN gap. The FOCUS is the Input gap, which means that PAM should look for the filler in the Input gap, and finally the filler should describe an appropriate plan for satisfying the hunger goal.

The second suggestion is,

```
Set TARGET = PLANNER of GOAL
To FOCUS = ACTOR
Using FOCUS-REQ
```

This says that PAM should look into the ACTOR slot of the input conceptualization for the filler for the PLANNER gap. FOCUS-REQ is a

test that says that if the FOCUS is not empty its value should be moved to the TARGET. This suggestion gets activated immediately and at the end of processing the first sentence PAM's memory has one conceptualization and one active request/suggestion.

```
(S-HUNGER    PLANNER: HUM0
              SOURCE: THEME = HUNGER-DRIVE
              PLAN: ( ))
```

and

```
(Suggestion:  Set TARGET = PLAN
              To FOCUS = !INPUT!
              Using SUITABLE-PLAN-RULE)
```

The next sentence is expressed as the following conceptualization,

```
((⇔ ($RESTAURANT CUSTOMER HUM0 RESTAURANT
    ORG0))))
```

This is an Action that says that the HUM0 did the restaurant Script (denoted by the prefix \$), that is, he ate at a restaurant. PAM has one active expectation that is looking for a plan, but what it gets is an Action. It goes into the Bottom-up mode, and retrieves the following rule or request. The rule says that "if someone eats in a restaurant, he must have had a plan to eat in a restaurant".

DO-RESTAURANT-PLAN RULE

```
Condition:  (( ⇔ ($RESTAURANT
                  CUSTOMER HUM0
                  RESTAURANT ORG0))))
```

```
Action:    ADD PLANBOX D0-$RESTAURANT-PLAN
            PLANNER()    RESTAURANT()
            ACTIONS()    SUBEPISODES()
```

Suggestions:

```
Set TARGET = PLANNER of PLANBOX
To FOCUS = CUSTOMER of script
Using FOCUS-REQ
```

```
Set TARGET = RESTAURANT of PLANBOX
To FOCUS = RESTAURANT of script
Using FOCUS-REQ
```

```
Set TARGET = ACTIONS
To FOCUS = !FOCUS!
Using FOCUS-REQ
```

PAM places the PLANBOX into the Input gap, and finds that it was expecting a plan. It incorporates this new structure into the story representation by means of the request that was active and looking for a plan. It then activates the three requests shown as suggestions. Two of these use FOCUS-REQ to find their data—the planner and the restaurant—and move it to the respective gaps. The action gap is filled by the pattern that triggered the rule. The remaining gap for SUB-EPISODES is unfilled and PAM will wait for more sentences that it will try and fit into this gap and others that may be created in the process. At this point, the story representation it has created is shown in Figure 14.17 below.

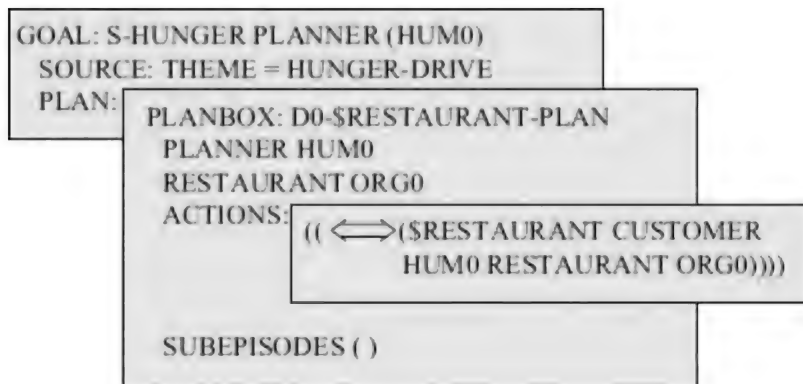


FIGURE 14.17 The Story reconstructed by PAM. There is an unfilled gap for SUBEPISODES. PAM would attempt to connect any subsequent sentences to this gap.

The ability of a program to understand a story, in the sense of being able to answer questions about it, depends upon the knowledge it has. The more the number of Scripts a system has, the more the number of stereotyped situations it will be able to recognize quickly. The more the number of Goals, Plans, Actions and the relations between them it knows, the more diverse the stories in which the program will be able to connect sentences by coherently. Schank and Abelson (1977) have identified the following different kinds of goals.

- Satisfaction goals like S-HUNGER, S-SEX and S-SLEEP. These are recurring biological needs of humans.
- Enjoyment goals like E-TREKKING, E-MUSIC and E-BRIDGE. Activities that are optionally pursued by people for enjoyment.
- Achievement goals like A-SKILL, A-GOOD-JOB and A-POSSESSIONS. These are long-term goals that someone may want to achieve.
- Preservation goals like P-HEALTH and P-REPUTATION. These are goals that crop up when something is threatened.
- Crisis goals are special class of P-goals that might need drastic action. If other goals are present then C-goals might take the highest precedence.
- Instrumental goals are goals that realize the precondition of other

goals. For example, if one has the goal of hitting someone, an instrumental goal of being near them might crop up.

- Delta goals are goals whose achievement results in state change. Examples are D-KNOW the goal of knowing something, D-PROX the goal of being in proximity of, and D-CONT the goal of getting control of something.

The means of achieving a Goal is to have a Plan. A Plan is made up of Actions and embodies a strategy for achieving the Goal. Associated with each type of Goal may be a set of Plans, called planboxes in PAM. A planbox is essentially a canned Plan. Once you deploy a planbox, you have made the intentions of your Actions clear.

Consider the goal D-CONT of getting control or possession of something. One assumes that the desired object is in possession of someone else. For example, "Moe wanted Calvin's toy truck" would translate to,

GOAL: (*DCONT* PLANNER HUM0 OBJECT PHYS0 OWNER HUM1 RECIPIENT HUM0)

Here HUM0, HUM1 and PHYS0 are tokens pointing to the appropriate frames representing the individuals and the object. PAM explains D-CONT goals by the themes that people want things either because they are fond of them, or they have a use for them.

Once PAM accepts the goal, it will set up expectations, looking for plans to achieve the goal. PAM associates several plans with achieving the D-CONT goal, and considers them in the following order. Their meanings are self-evident.

1. ASK
2. EXPLAIN
3. BARGAIN
4. THREATEN
5. OVERPOWER
6. STEAL

Associated with each planbox is the possible set of actions. If PAM hears next that "Moe walked over to Calvin", it would go into a Bottom-up mode and figure out that walking over must be part of some plan Moe had to get control of Calvin's truck.

PAM already realizes that there is a conflict between Moe wanting the truck and Calvin wanting to keep it, and would know that the conflict could be resolved in any way.

Let us say Moe ASKs Calvin to hand over the truck and Calvin refuses. PAM would be expecting this to be a possibility and will explain that this is because Calvin wanted to keep his truck. If at some point in the story Moe were to THREATEN Calvin then PAM would know that Calvin has some more difficult options. If Calvin yields and hands over the truck, PAM would understand that he had conflicting goals—of holding on to his truck or preserving his health—and had chosen the more prudent option.

PAM needs to know the relations between goals, plans and actions to understand stories. One can make things more interesting by creating characters of different types that use different planboxes in some order, for example bullies or honest people. Figure 14.18 shows some possible personality types with respect to the D-CONT goal.

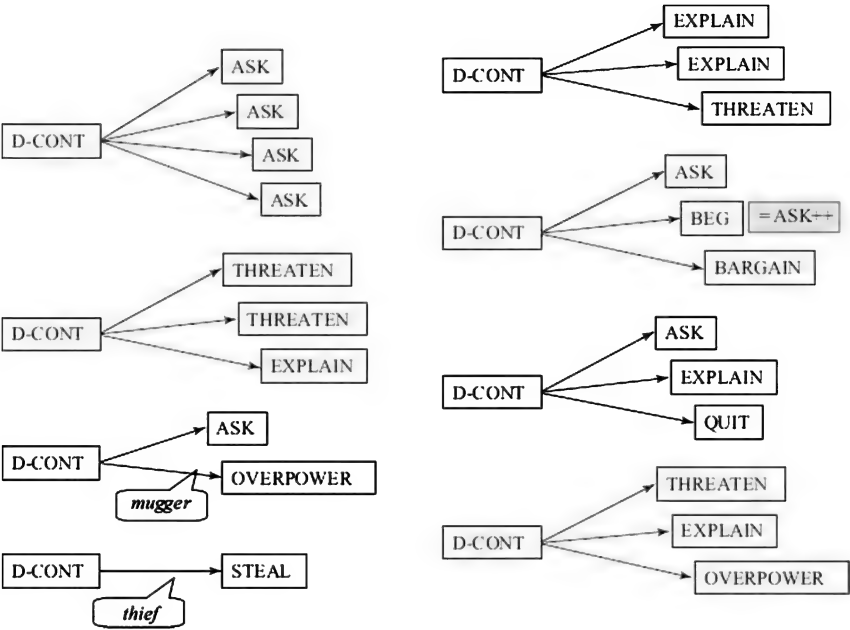


FIGURE 14.18 Different personality types may use different sequences of plans to try and achieve a D-CONT goal. The reader is invited to provide the other labels.

To be able to understand stories with many goals and characters, PAM has a model of goal interactions.

- 1. Goal Subsumption** When different goals of a person have a positive interaction. For example “Anastasia went to Besant Nagar to book her tickets. While there, she also spent time on the beach and had her dinner at an eatery.”
- 2. Goal Conflict** When different goals of a character compete for resources. For example “Ayesha wanted to go to the beach but she had to submit an assignment the next day.”
- 3. Goal Competition** An inimical relationship between the goals of different characters. For example, “Ayuta wanted to watch a soccer game but Akello insisted on watching the movie.”
- 4. Goal Concord** When goals of different characters have a positive interaction. For example, “Amina wanted to get rid of her iPod. She sold it

to Akna who was looking for a second-hand one."

Recognizing these situations enables PAM to answer questions like *"Why did Calvin give the truck to Moe?"* with *"Because he didn't want to get hurt."*

14.5.3 Story Writing

Story understanding is the task of making sense of a sequence of input sentences. PAM uses its knowledge of goals and plans to link the different sentences in the input story.

Story generation can be seen as the inverse task which can work as follows.

1. Create a cast of characters. For each character, ascribe a set of planboxes for each type of goal
2. Create a set of props.
3. Create a set of goals for some of the cast.
4. Create a sequence of actions by choosing plans for each character that has a goal. Induce more goals in the process.
5. Output the final story in a natural language.

One such program called TALE-SPIN was written by James Meehan from the Yale group as part of his thesis in 1976 (Meehan, 1981). TALE-SPIN started with an initial set of S-goals and proceeded to generate a story. The program worked in an interactive fashion, asking the user to make certain choices. For example it could start with the sentence "One day Sam was very hungry". Sam, being a bear, is known to like honey, and TALE-SPIN would create a D-CONT goal for Sam, which would link to a D-KNOW goal in which Sam wants to know where some honey is. If the initial settings said that Betty bee had some honey then the program would create a goal for Sam wanting to get the honey from Betty.

At this point, TALE-SPIN would ask the user how honest Sam was, and whether Sam thought Betty was vain, and whether Sam liked Betty. Now if Sam was honest and liked Betty, he could not possibly THREATEN her, and since she was not vain, flattery would not work either. Sam decides to persuade her by means of a falsehood, and TALE-SPIN creates the goal,

"Sam wanted Betty to fly away from her beehive"

We will not go into the details here but in the story generated by TALE-SPIN, Sam decides that if he tells Betty where a flower is, she might go off to the flower (and therefore not be at the beehive). So Sam goes over to Betty and tells her about the flower, and Betty being famished goes off to the flower, and Sam gets his opportunity to help himself to some honey from Betty's beehive.

Generating the events and states that make up the story is only the first step. Converting the output of TALE-SPIN into an enchanting tale requires rhetorical and language skills. The storyteller needs to decide

what conceptualizations to choose to convey, and how to embellish the linguistic output to avoid it being monotonous (see (Mann and Thompson, 1988), (McKeown, 1992), (Hovy, 1993), (Somayajulu, 1998)). That is beyond the scope of this book. We only illustrate the *need* for doing so by looking at an excerpt generated by TALE-SPIN (Meehan, 1981). The words in the original output were all in upper case. Observe that the *action* begins only in the second paragraph!

"Once upon a time John bear lived in a cave. John knew that John was in his cave. There was a beehive in a maple tree. Tom bee knew that the beehive was in the maple tree. Tom was in his beehive. Tom knew that Tom was in his beehive. There was some honey in Tom's beehive. Tom knew that the honey was in Tom's beehive. Tom had the honey. Tom knew that Tom had the honey. There was a nest in a cherry tree. Arthur bird knew that the nest was in the cherry tree. Arthur bird was in his nest. Arthur knew that Arthur was in his nest. Arthur knew that John was in his cave. John knew that Arthur was in his nest. John knew that Tom was in his beehive. There were some boysenberries near a bush. There was a lily flower in a flowerbed. Arthur knew that the boysenberries were near the bush. John knew that the lily flower was in the flowerbed.

One day John was very hungry. John wanted to get some honey. John wanted to find out where there was some honey. John liked Arthur. John wanted Arthur to tell John where there was some honey. John was honest with Arthur. John wasn't competitive with Arthur. John thought that Arthur liked him. John thought that Arthur was honest with him. John wanted to ask Arthur whether Arthur would tell John where there was some honey. John wanted to get near Arthur. John walked from a cave exit....¹⁸

14.5.4 BORIS

The high point of the work on knowledge structures for natural-language comprehension at Yale was probably the program BORIS written by Michael Dyer (1983), (Lehnert et al, 1983). BORIS incorporated all the structures explored by his predecessors in addition to some designed by Dyer. BORIS had deep knowledge about its somewhat narrow domain of discourse, and that is reflected in the way it answers questions. Another interesting feature of BORIS was that it combined language parsing with memory search. It searched its memory as it read each word, and thus combined a bottom flavour with the top-down memory based one. A big advantage of integrating memory search with language parsing is that the two processes can influence each other, thus making tasks like word sense disambiguation easier. When BORIS is listening to a question, it is not only constructing a Conceptual Dependency representation, but also consulting its episodic memory looking for the answer, and often knows the answer even before completely parsing the question. Here is an example from (Dyer et al., 1981),

“George was having lunch with another teacher and grading homework assignments when the waitress accidentally knocked a glass of coke on him. George was very annoyed and left, refusing to pay the check. He decided to drive home to get out of his wet clothes.

When he got there, he found his wife Ann and another man in bed. George became extremely upset and felt like going out and getting plastered...”

Q: *Why didn't George pay the check?*

BORIS: *Because the waitress spilled coke on him.*

Q: *How did Ann feel when George caught her cheating on him?*

BORIS: *She was surprised.*

The architecture of BORIS is shown in Figure 14.19.

The most significant new structure introduced by Dyer was the Thematic Abstraction Unit (TAU). The TAU is an abstraction of a planning situation where some expectation failure occurs in planning. A TAU is designed to capture the kind of distilled experience that is often expressed in adages like “the pot calling the kettle black”, “throwing stones when you live in a glass house”, or “closing the barn door after the horse has escaped”. A TAU represents an abstract situation-outcome patterns in situations where plans are deployed in terms of (1) the plan used, (2) its intended effect, (3) why it failed, and (4) how to avoid or recover from that type of failure. The TAU for the “closing the barn door” adage is represented as (Dyer, 1983),

TAU-POST-HOC

- (1) x has a preservation goal G active, since enablement condition C is unsatisfied.
- (2) x knows a plan P that will keep G from failing by satisfying C
- (3) x does not execute P and G fails.
 x attempts to recover from the failure of G by executing P .
 P fails since P is effective for C , but not in recovering from G 's failure.
- (4) In the future, x must execute P when G is active and C is not satisfied.

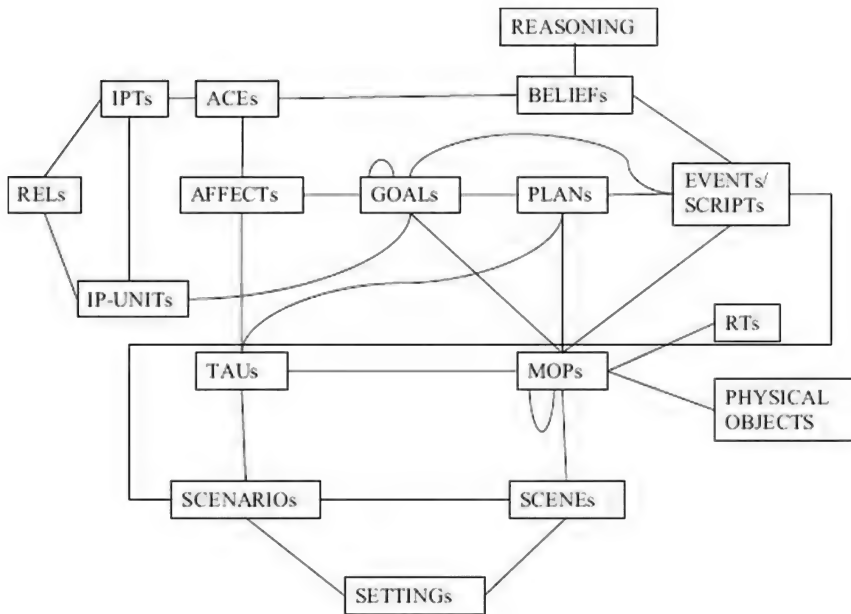


FIGURE 14.19 Knowledge interactions and dependencies between knowledge structures in BORIS (Dyer, 1983).

Another example of a TAU is when someone does something inadvertently to upset a friend and does something to make up for it.

TAU-REG-MISTAKE

x (the schlemiel) causes an unintended event E

E motivates a preservation goal G on the part of y (the schlimazel)

If x and y have a positive interpersonal relationship

Then x is motivated to serve as an agent for y in recovering G

Associated with TAUs are AFFECT related expectations that talk of emotional responses of people. In this example, the expectations are,

If x and y have a positive interpersonal relationship

then x will feel regret, guilty, embarrassment, etc.

y will feel upset and angry at x .

Dyer reports that the above TAU is applied by BORIS on the following passage:

“Richard spilled a cup of coffee on Paul. Paul seemed very annoyed by this so Richard offered to drive him home for a change of clothes.”

An AFFECT is a knowledge structure to capture the emotional responses of people to the outcomes of plans. An AFFECT has six basic constituents.

1. **STATE** Characterizes the nature of the sentiment and can be positive (POS) or negative (NEG).
2. **CHAR** Identifies the actor in the narrative who is feeling the emotion.
3. **G-SITU** Refers to the goal situation that is the context for the emotion.
4. **TOWARD (optional)** Identifies if the emotion is directed towards someone.
5. **SCALE** Measures the intensity of the emotion. BORIS supports two values: >NORM (more than normal) and <NORM (less than normal).
6. **E-MODE (optional)** If the characters have an expectation about likely future outcomes.

Some examples of AFFECTs are,

Happy, joyous, glad	(AFFECT STATE (POS) CHAR x G-SITU (goal of x achieved))
Grateful, thankful	(AFFECT STATE (POS) CHAR x G-SITU (y caused (goal of c achieved) to occur) TOWARD y)
Fearful, worried	(AFFECT STATE (NEG) CHAR x G-SITU (P-goal is active) E-MODE (UNEXPECTED))
Guilty, ashamed, Embarrassed	(AFFECT STATE (NEG) CHAR x (G-SITU (goal of y thwarted by x)) TOWARD y)

Emotions arise not only because of what happens to you, but also due to what happens to other people. In BORIS, interpersonal themes (IPTs) are structures that capture this relationship, expressed in an empathy table expressing rules such as the following:

- em1: If x and y are friends
and y has a goal failure
Then x will experience a NEG AFFECT.
- em2: If y has a failure
and x experiences POS AFFECT
Then either x and y are enemies,
or they are in conflict over this goal.

BORIS also employs a structure called ACE (Affect as a Consequence of Empathy) when a character expresses empathy for another character. Some examples of ACEs are,

commiserate,	theme = (IPT-FRIENDS x y)
condole	x MTRANS TO y that [goal failure (y) causes: x feel NEG]
felicitate,	theme = (IPT-FRIENDS x y)
congratulate	x MTRANS TO y that [goal success (y) causes: x feel POS]
gloat	theme = (IPT-ENEMIES x y) x MTRANS TO y that [goal failure (y) causes: x feel POS]
envy,	theme = (IPT-ENEMIES x y)
jealous,	x MTRANS TO y
spiteful	that [goal success (y) causes: x feel NEG]

Going through the literature on BORIS, one realizes that it is a humongous representation and programming exercise, and Dyer has had to improvise and create knowledge structures in a somewhat ad-hoc manner. It clearly brings out the need for large amounts of knowledge that need to be encoded for a software system to display the kind of breadth, depth and versatility in handling language, and what is commonly known as common-sense knowledge. In the somewhat nebulous classification of AI research into *neats* vs *scruffies*¹⁹, BORIS probably falls into the latter class. Another project that ventured into representing vast amounts of common-sense knowledge was the project CYC lead by Doug Lenat (1995).

Echoing what Charniak observed in his Ms. Malaprop paper, Dyer says in his thesis that if BORIS is to understand why Richard makes the offer of driving him for a change of clothes to Paul, it must know mundane (common sense) facts like what “spilling” means, how liquids affect clothes, that people don’t like to be uncomfortable, where clothes are kept, etc.

We will depart from this study of (handcrafted) knowledge structures with the observation that much still needs to be done, if we are to build computer programs that are as knowledgeable as the common man on the street in everyday conversation.

14.6 Inheritance in Taxonomies

While constructing taxonomies, one is faced with a trade-off between economy of expression and the universal nature of statements. If we want all our statements to be universally true, we will need to build much larger semantic networks, because universal statements do not allow exceptions. Economy of expression, on the other hand, demands that we

make statements that are *generally* true, but which might admit exceptions. Economy is the motivation for ascribing properties to super classes and inheriting them in the subclasses. Exceptions are taken care of by overriding the inherited default values with explicitly stated ones.

In this section, we take a closer look at the kind of reasoning that goes into inheritance, specially when there is conflict in the values being inherited from different ancestors. The question is: Given that a slot (or property) can inherit multiple fillers (or values), which one does one choose to accept?

In the discussion that follows, we will adopt the following stylized notation that is commonly used. A node will be referred to by a single word that either names the instance, or the class, or names the filler (or value) being inherited. Further, for the sake of discussion, we will only talk about a property value and its negation. For example, instead of choosing between “leaves are green” and “leaves are orange”, we will choose between “leaves are green” and “leaves are not green”. This means that the question we are asking is a binary one of the type “Is A a kind of B?” or “Does A have property P?”. The answer is encoded in a directed edge from A to B or P. A positive link signifies *yes* and a negative link *no*. Thus, we imagine that the network being displayed has been custom made for the question we are asking.

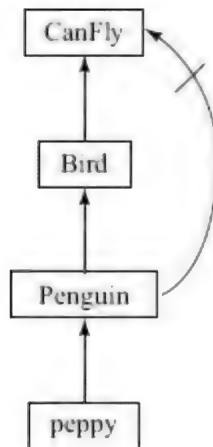


FIGURE 14.20 An inheritance network. The node labelled *CanFly* represents the class of things that “can fly”. The edge with a dash across represents the fact that that property is *not* inherited.

Consider the following set of statements:

- Peppy is a penguin.
- Penguins are birds.
- Birds can fly.
- Penguins cannot fly.

We represent the sentences as the graph in Figure 14.20. We adopt the notation of representing the edges by the two nodes separated by a dot. If the relation is a negative one, as in the last sentence, we include

the negation sign. The graph can then be represented by the edges,

peppy · Penguin
Penguin · Bird
Bird · CanFly
Penguin · ¬CanFly

The question we want to ask is whether “Peppy can fly” is true, which is represented by the statement “ $\text{peppy} \rightarrow \text{CanFly}$ ”. In general, the inheritance question can be answered by traversing the inheritance network upwards from the node in question to check whether one can reach the desired property. In this example, there are two paths that one can find from “peppy” to “CanFly”. One (peppy·Penguin·Bird·CanFly) is a positive path, implying that Peppy can inherit that property. The other (peppy·Penguin·¬CanFly) is a negative path, implying that it cannot. Which one is more acceptable?

One heuristic would be to accept the shortest path (from the set of available paths). In this example, this would work because the shorter path does indeed represent the intended conclusion that (while birds in general can fly) penguins cannot fly, and since Peppy *is* a penguin, it is reasonable to infer that it cannot fly.

But what if the two paths to a node are of equal length? For example, we can modify our set of sentences so that the network looks as the one on the left in Figure 14.21. In that case, we do not have any reason for preferring either of the two possible paths.

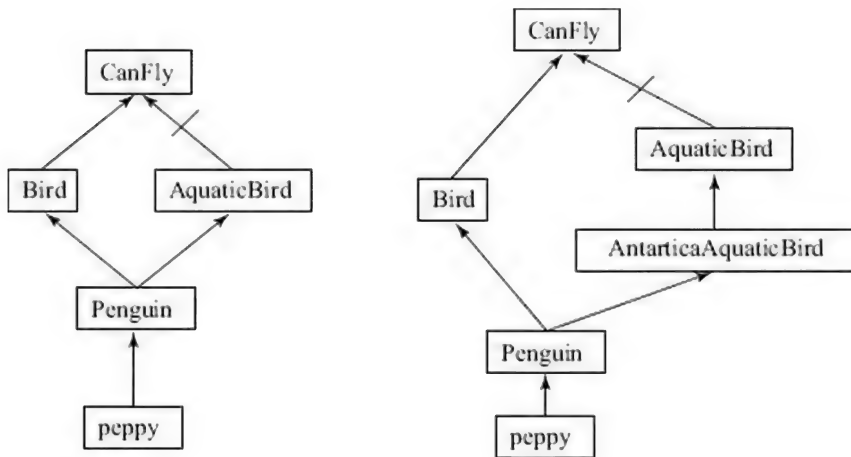


FIGURE 14.21 The network on the left has equal paths, and hence no preferred inference. Adding one more node in the negative path now makes it longer. The shortest path heuristic would conclude that Peppy can fly.

The fact that we could add another sentence and change the decision does not bode well for the shortest path heuristic. In fact, we can add one more node to the network as shown on the right in the figure, and our

original conclusion is reversed. Now the shorter path says that Peppy can fly. The only thing we did was to add some more information in the form of class subclass relations to the given set. The shortest path heuristic is oblivious to the fact that adding more detailed information does not change the nature of nodes. Clearly, we need a more robust method for choosing what properties to inherit.

Before we proceed, we must observe that only one kind of negative path is acceptable—which has *exactly one* negative edge as the *last* edge in the path. That is, the path can only be of the form

$$N_1 \cdot N_2 \cdot \dots \cdot N_j \cdot \neg N_k$$

In essence, we cannot add an edge *after* a negative edge. Consider the following invalid paths,

1. peppy·Penguin· \neg Hexapoda·LivingCreatures
2. peppy·Penguin· \neg Hexapoda·Arthropoda

The first one says that penguins are not hexapoda, and hexapoda are living creatures. The second one says that penguins are not hexapoda, and that hexapoda are a kind of arthropoda. Now penguins *are* living creatures, but they *are not* arthropoda. But both the paths have the same structure. This shows that a path that has edges *after* the negative edge cannot be trusted, and is therefore invalid.²⁰ Therefore, if a negative edge is there in a path, it must be the last edge in the path, and the path must end with it.

The following discussion on admissible paths is derived from (Stein, 1989; 1992) and (Brachman and Levesque, 2004). A path supports a conclusion in one of the following two ways:

- $A \cdot N_1 \cdot \dots \cdot N_k \cdot X$ supports the conclusion $A \rightarrow X$
- $A \cdot N_1 \cdot \dots \cdot N_p \cdot \neg X$ supports the conclusion $A \rightarrow \neg X$

An inheritance graph Γ supports the conclusion $A \rightarrow X$ (or $A \rightarrow \neg X$), if there exists an *admissible* path $A \cdot N_1 \cdot \dots \cdot N_k \cdot X$ (or $A \cdot N_1 \cdot \dots \cdot N_p \cdot \neg X$) in Γ .

A path $A \cdot N_1 \cdot \dots \cdot N_p \cdot (\neg)X$ is *admissible*, if every edge in it is *admissible with respect to A*.

An edge $V \cdot (\neg)X$ is *admissible in Γ with respect to A*, if there is a *positive path* $A \cdot N_1 \cdot N_2 \cdot \dots \cdot N_r \cdot V$ ($r \geq 0$) in the graph, and

1. each edge in $A \cdot N_1 \cdot N_2 \cdot \dots \cdot N_r \cdot V$ is *admissible in Γ with respect to A*,
2. no edge in $A \cdot N_1 \cdot N_2 \cdot \dots \cdot N_r \cdot V$ is *redundant in Γ with respect to A*,
3. no intermediate node in $A \cdot N_1 \cdot N_2 \cdot \dots \cdot N_r$ is a *preemptor* of the edge $V \cdot (\neg)X$

An *admissible edge* then is one that has an *admissible path* with no *redundant edge* in the path leading up to it, and its conclusion is not preempted by any node in the path.

A node P in a path $A \cdot \dots \cdot P \cdot \dots \cdot V$ is a *preemptor of $V \cdot X$* with respect to A , if $P \cdot \neg X$ belongs to the graph Γ . Likewise, P is a *preemptor of $V \cdot \neg X$* with respect to A if $P \cdot X$ is in Γ . Essentially, a *preemptor* provides a shorter path to the opposite conclusion.

A positive edge $B \cdot W$ is redundant in Γ with respect to A , if there is a path $B \cdot L_1 \cdot L_2 \cdot \dots \cdot L_k \cdot W$ for which,

1. each edge in $B \cdot L_1 \cdot L_2 \cdot \dots \cdot L_k \cdot W$ is admissible in Γ with respect to A , and
 2. there is no C and an edge of the form $C \cdot \neg L_j$ or $C \cdot \neg W$ that is admissible with respect to A in Γ
- Consider the inheritance graph shown in Figure 14.22.

The following edges are inadmissible w.r.t. a . Edge $c \cdot d$ is not admissible because it is preempted by node b . Edges $f \cdot d$, $f \cdot g$, $g \cdot h$, $e \cdot f$, $a' \cdot f$, $a' \cdot e$ and $e \cdot g$ are not admissible because they do not have a positive path from a leading up to them. Note that $d \cdot h$ is admissible and is not preempted by c because the path to d is $a \cdot b \cdot d$.

With respect to a' , the following edges are inadmissible. Edge $f \cdot g$ is inadmissible because on the non-redundant positive path, $a' \cdot e \cdot f$ node e is a preemptor for g . The other path $a' \cdot f \cdot g$ has the redundant edge $a' \cdot f$. The edge $g \cdot h$ is inadmissible because not all edges (in particular edge $f \cdot g$) on the positive nonredundant positive path to g are admissible. Edges $a \cdot b$, $b \cdot c$, $b \cdot f$, $b \cdot d$, $c \cdot d$, $c \cdot h$ are inadmissible because they do not have a positive path leading up to them from a' . Note that $f \cdot d$, $d \cdot h$, have a nonredundant path via $a' \cdot e \cdot f$ and are admissible with respect to a' .

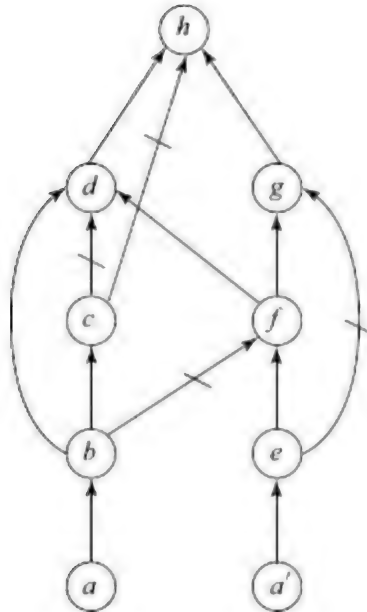


FIGURE 14.22 A sample inheritance network.

14.6.1 Extensions

Given a possibly ambiguous inheritance graph, one might be interested in

knowing a consistent set of properties that can be inherited by a node, or a set of conclusions that one can draw with respect to a node. Given a node A , we define a *credulous extension* of an inheritance graph Γ as a *maximal consistent* subgraph of Γ that represents a maximal set of *consistent* conclusions with respect to node A . A subgraph is consistent if it does not support both $A \rightarrow X$ and $A \rightarrow \neg X$. A subgraph is maximal if one cannot add any edge to the graph while maintaining the consistency property.

The three sub-graphs shown in Figure 14.23 are the three possible credulous extensions of the inheritance graph from Figure 14.22, with respect to node a in the graph. As one can see, they differ mostly on whether a can be concluded to be d and h or not. While the extension 1 allows both, extension 2 allows the conclusion that $a \rightarrow d$ but not $a \rightarrow h$ (it allows $a \rightarrow \neg h$), and extension 3 allows neither d nor h .

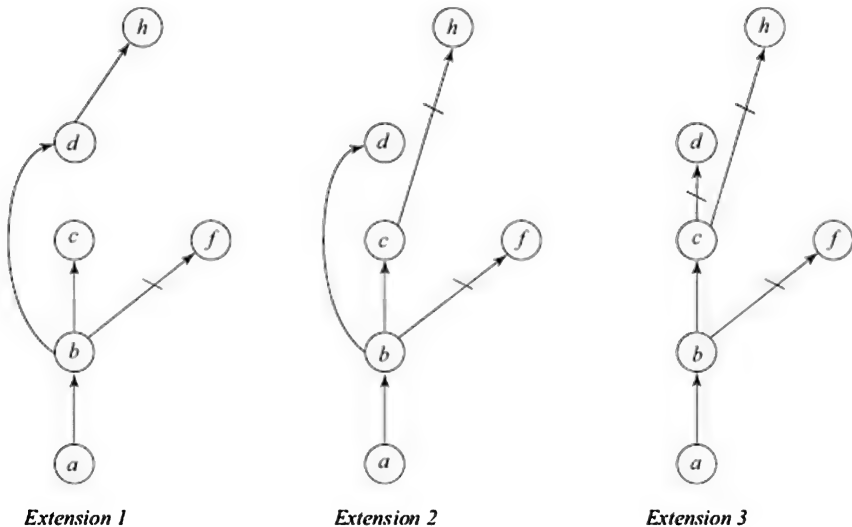


FIGURE 14.23 The three credulous extensions of the graph in Figure 14.22 for node “a”. Extensions 1 and 2 are preferred over extension 3 because 3 includes the edge $c \rightarrow d$ that is not admissible in the original network. Between extensions 1 and 2, neither is preferred over the other.

One might ask whether one might prefer one extension over another. Looking at the above three extensions, we can see that all the edges in extensions 1 and 2 are admissible in the original graph, but extension 2 contains an edge $c \rightarrow \neg d$ which was not admissible in the original graph. We say that the first two extensions are preferred over the third.

A credulous extension E is preferred to a credulous extension F with respect to a node A , if F contains an edge $X \rightarrow Y$ (or $X \rightarrow \neg Y$) that is inadmissible with respect to A in the original inheritance graph and E does not contain that edge, but agrees with F up to node X . A credulous extension is a preferred extension, if there is no other credulous

extension that is preferred to it.

Finally, given an inheritance network Γ , what can we infer about a given node A ? Various forms of reasoning have been proposed (Brachman and Levesque, 2004).

Credulous Reasoning Allow all conclusions that are supported by *any* preferred extension.

Skeptical Reasoning Believe all those conclusions X that are supported by $A \cdot N_1 \dots N_k \cdot X$ in *all* the preferred extensions.

Ideally Skeptical Reasoning Believe all those conclusions $A \rightarrow X$ that are supported by some path in *each* preferred extension.

14.7 Description Logics

The taxonomies built in frames, semantic nets and MOPs are all hand crafted, and do not necessarily have a logical basis. The inferences we make about inheritance of property values are only plausible inferences. In this section, we explore the representation of compound concepts in terms of simpler ones, using what are known as *Description Logics*. In a Description Logic (DL), the property values of a compound concept are derived from their descriptions and from the properties of atomic concepts.

One of the earliest approaches to devise a compositional knowledge representation scheme was by Gottfried Leibniz when he introduced the notions of *characteristica universalis* and *calculus ratiocinator* in 1679 (see for example (Jaenecke, 1996)). Like most philosophers who were investigating the principles of thought, Leibniz was also trying to establish a rational basis of human thought. His basic principles are.

1. All our ideas are compounded from a very small number of simple ideas, which form the “alphabet of human thought”.
2. Complex ideas are formed by simple ideas by “uniform and symmetrical combination”, analogous to arithmetical multiplication.

At one point, Leibniz explored the possibility of representing all concepts using prime numbers. For example if grey = 3, elephant = 7, inBangalore = 11, isBookstore = 23 then the product 231 represents a grey elephant in Bangalore. Then, every compound concept and individual could be a number. For example, the Strand bookstore in Bangalore might have a number 5981426. Given the number 5981426, we can check if the property “inBangalore” is true if it is divisible by 11, and the property “isBookstore” by checking if it is divisible by 23. If Clyde is assigned the number 6123278, we can determine that he is an elephant because 6123278 is divisible by 7, but is not grey because it is not divisible by 3. In general, given two concepts X and Y , we can find out if X is a subconcept of Y by checking if Y is divisible by X . One objection to this scheme would be that the assignment of prime numbers for

properties is one-dimensional, opaque and arbitrary, and lacks enough *description*.

While Leibniz did not publish much in philosophy, Bertrand Russell (1945) has said that he had in fact developed symbolic logic to a considerable degree.²¹ Modern Description Logics have the same compositional flavour, though the operations allowed for combination are more generalized and diverse.

There are a number of languages of varying expressivity and completeness in the family of Description Logics (see (Baader et al., 2003)). There has been considerable amount of work on Description Logics owing to their attractiveness as the basis of languages for defining meta data on the Web. We confine our study to a simple version described in (Brachman and Levesque, 2004). The main feature we want to explore is how taxonomies can be derived from descriptions, and how one can use the descriptions to answer questions relating to the abstraction hierarchy.

14.7.1 The DL language

The DL language, like the First-Order Language (FOL), also comprises two parts, the logical part, that is the core of the language, and the nonlogical part that pertains to the domain.

The vocabulary of logical symbols of DL is made up of the punctuation symbols “[”, “]”, “(“and “)”; the connectives “ \exists ”, “ \neq ”, and “ \rightarrow ”; the concept forming operators “ALL”, “EXISTS”, “FILLS” and “AND”; and the set of positive integers 1, 2, 3,....

The nonlogical symbols are of three types,

Atomic Concepts Atomic concepts, like “Girl”, “Student” which are basic concepts of the language. They define the primitives in terms of which other concepts are defined, for example “a school that has only girl students”. Atomic concepts are the simplest classes or categories, and are denoted by capitalized words.

Roles Roles signify relations between objects. For example, “:StudentOf” may be a relation between a “Person” and a “Discipline”. Roles are denoted by capitalized words preceded by a colon.

Constants Constants refer to elements in the domain, denoted by uncapitalized words, like aditi or book21.

The Description Logic DL is a language for describing concepts or subsets of the domain D . The simplest concepts are described by the atomic concepts. Compound concepts are combinations of atomic concepts and concept forming operators. The following are the ways of constructing concept descriptions.

- Every atomic concept is a concept;
- If r is a role and d is a concept then $[ALL\ r\ d]$ is a concept;

- If r is a role and n is a positive integer then $[\text{EXISTS } n \ r]$ is a concept;
- If r is a role and c is a constant then $[\text{FILLS } r \ c]$ is a concept;
- If d_1, d_2, \dots, d_n are concepts then $[\text{AND } d_1, d_2, \dots, d_n]$ is a concept.²²

Like in FOL, we can define the meaning of terms by means of an interpretation $\mathcal{I} = \langle D, I \rangle$ where D is a domain and I is a mapping from the DL to the domain. The elements of DL may be interpreted as follows.

- Every constant c maps to an element of the domain, $I[c] \in D$.
- Every atomic concept a maps to a subset of D , $I[a] \subseteq D$.
- Every role r maps to a binary relation on D , $I[r] \subseteq D \times D$.

The concepts constructed using the given operators correspond to subsets of the domain as follows.

- $[\text{ALL } r \ d]$ is the set of all elements in the domain that are related by r only to elements of type d .

$$I[\text{ALL } r \ d] = \{x \in D \mid \text{for all } y \text{ if } \langle x, y \rangle \in I[r] \text{ then } y \in I[d]\}$$

For example, $[\text{ALL} : \text{Grade A}]$ is the set of students who earn only A's, and $[\text{ALL} : \text{Daughter Bright}]$ is the set of people, all of whose daughters are bright.

- $[\text{EXISTS } n \ r]$ is the set of all those elements that are related to at least n elements by the relation r .

$$I[\text{EXISTS } n \ r] = \{x \in D \mid \text{there are at least } n \text{ distinct } y \text{ such that } \langle x, y \rangle \in I[r]\}$$

For example, $[\text{EXISTS } 1 : \text{Daughter}]$ is the set of all those (lucky) people who have at least one daughter, and $[\text{EXISTS } 2 : \text{Friend}]$ is the set of all those people who have at least 2 friends.

- $[\text{FILLS } r \ c]$ is the set of all those elements that are related by r to c .

$$I[\text{FILLS } r \ c] = \{x \in D \mid \langle x, I[c] \rangle \in I[r]\}$$

For example, $[\text{FILLS} : \text{FanOf gerrard}]$ is the set of all fans of Gerrard, and $[\text{FILLS} : \text{Study ai}]$ is the set of people who study AI.

- $[\text{AND } d_1, d_2, \dots, d_n]$ is the set of all those elements that are of the type d_1 and d_2, \dots and d_n .

$$I[\text{AND } d_1, d_2, \dots, d_n] = I[d_1] \cap I[d_2] \cap \dots \cap I[d_n]$$

For example, $[\text{AND Student Girl } [\text{FILLS} : \text{Study ai}]]$ is the set of all girl students who study AI.

Figure 14.24 illustrates the composition of new concepts using the four concept forming operators. The arrows represent the roles. Elements shaded black are the ones belonging to the new concept.

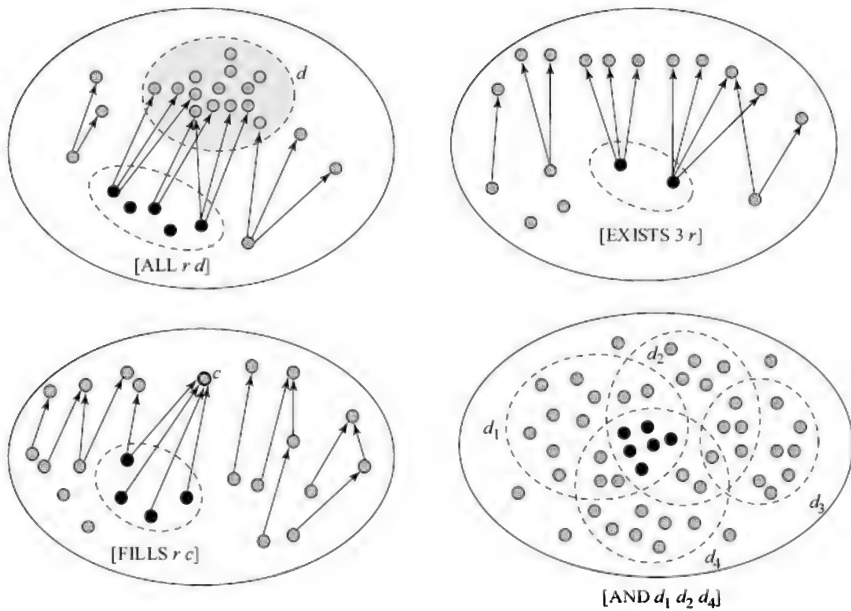


FIGURE 14.24 The four concept forming operators. The arrow represents the relation r . The members of the set or concept formed are shown in black filled circles.

Given a set of atomic concepts and roles, one may define complex concepts using the concept forming operators. For example, we could define a class of films that has at least 2 villains, 4 songs, and all whose actors are more than 12 years old and live in Mumbai, as follows:

```
[AND  Film
    [EXISTS 2 :Villain]
    [EXISTS 4 :Song]
    [ALL :Actor [AND [FILLS :MinAge 12]
                     [FILLS :LivesIn mumbai]]]]
```

The sentences of DL assert the *Is-A* and *Instance-Of* relations described in the beginning of the chapter. The *Is-A* relation is expressed using " \sqsubseteq " and the *Instance-Of* using " \rightarrow ". In addition, DL uses " \doteq " to express the fact that two concepts are equivalent. The following are the sentences of the language DL,

- If d_1 and d_2 are concepts then $(d_1 \sqsubseteq d_2)$, read as d_1 is subsumed by d_2 , is a sentence. The sentence is true if $I(d_1) \subseteq I(d_2)$.
- If d_1 and d_2 are concepts then $(d_1 \doteq d_2)$, read as d_1 is equivalent to d_2 , is a sentence. The sentence is true if $I(d_1) = I(d_2)$.
- If c is a constant and d is a concept, then $(c \rightarrow d)$, read as c satisfies d , is a sentence. The sentence is true if $I(c) \in I(d)$.

The equivalence sentence can be used to define and give names to new concepts. For example, we might want to name the class of films

defined above as *BFormulaFilm*. We can do so by the statement,

```
BFormulaFilm ≐ [AND Film
                  [EXISTS 2 :Villain]
                  [EXISTS 4 :Song]
                  [ALL :Actor [AND [FILLS :MinAge 12]
                                   [FILLS :LivesIn Mumbai]]]]
```

The equivalence statement is a short form for two subsumption statements. The sentence $(d_1 \doteq d_2)$ is equivalent to writing $(d_1 \sqsubseteq d_2)$ and $(d_2 \sqsubseteq d_1)$.

In FOL, the above statement would be written as,

```
∀x (BFormulaFilm(x) ≡ (Film(x) ∧
                       ∃v1v2 (v1 ≠ v2 ∧ Villian(x, v1) ∧ Villian(x, v2)) ∧
                       ∃s1s2s3s4 (AllDistinct(s1, s2, s3, s4) ∧ Song(x, s1) ∧
                       Song(x, s2) ∧ Song(x, s3) ∧ Song(x, s4)) ∧
                       ∀z (Actor(x, z) ⊃ (MinAge(z, 12) ∧ LivesIn(z, Mumbai))))
```

Observe that there are no variables in DL expressions, only predicate names that stand for categories and binary relations, and constants from the domain. A predicate name like “Film” is used here to stand for all objects in the domain that are films.

Reasoning about concepts or predicates is called *terminological reasoning* and constitutes the *TBox* of a DL, while reasoning about constants is called *assertional reasoning* and constitutes the *ABox* of a DL system.

The domain or the universe of discourse is represented by the concept *Thing*. Every element satisfies *Thing* and every concept is subsumed by *Thing*.

If d is a concept, $(d \sqsubseteq \text{Thing})$ is *true*, and

If c is a constant, $(c \rightarrow \text{Thing})$ is *true*.

14.7.2 Inheritance

The two questions we are interested in asking are related to inheritance. They are, does a given element satisfy a given class (or concept), and, whether a given class is subsumed by another class. In FOL, we could show subsumption by a process of chaining inferences. For example, given that

```
Dog(fido25)
∀x (Dog(x) ⊃ Animal(x))
∀x (Animal(x) ⊃ LivingCreature(x))
```

We could conclude by a deductive theorem proving process that,

```
LivingCreature(fido25), and
∀x (Dog(x) ⊃ LivingCreature(x))
```

The equivalent (given) statements in DL are,

$\text{fido25} \rightarrow \text{Dog}$
 $\text{Dog} \sqsubseteq \text{Animal}$
 $\text{Animal} \sqsubseteq \text{LivingCreature}$

And the sentences we want to show to be true are,

$\text{fido25} \rightarrow \text{LivingCreature}$
 $\text{Dog} \sqsubseteq \text{LivingCreature}$

The approach taken in DL is quite different. First, observe that if we can show that a dog is a living creature, we can also show that Fido is a living creature. So the key thing is to be able to show that subsumption holds between the two concepts. In DL, we show subsumption by *comparing the descriptions* of the two concepts in question. If an animal is defined as a living creature with some additional properties,

$\text{Animal} \doteq [\text{AND LivingCreature } X_1 X_2 \dots X_k]$

and a dog is defined as an animal with some additional properties,

$\text{Dog} \doteq [\text{AND Animal } Y_1 Y_2 \dots Y_p]$

then we can effectively write,

$\text{Dog} \doteq [\text{AND LivingCreature } X_1 X_2 \dots X_k Y_1 Y_2 \dots Y_p]$

This says the set of dogs (or Dog concept) is the *intersection* of the set of living creatures (the LivingCreature concept) and some other sets. The set of dogs then is *necessarily* a subset of the set of living creatures which holds iff ($\text{Dog} \sqsubseteq \text{LivingCreature}$).

This is a process of *structure matching*. We only compare the two descriptions after reducing them to a canonical form to decide whether one concept subsumes another or not. The comparison process will be linear in the length of the descriptions.

14.7.3 Normalization

The process of reducing DL expressions into a canonical form is called *normalization*. In the canonical form, any names assigned to compound concepts are replaced by their equivalent descriptions, the description contains only atomic concept names, and any redundant expressions are removed. The following are the replacement steps in the normalization procedure. Replace,

- any concept n by its definition d , if there is a statement of the form $n \doteq d$,
- an expression of the form

[AND ... X Y [AND P Q] ... R T] with
[AND ... X Y P Q ... R]

- an expression of the form

[AND ... [ALL r d₁] ... [ALL r d₂] ...] with
[AND ... [ALL r [AND d₁ d₂]] ...]

- an expression of the form

[AND ... [EXISTS n₁ r] ... [EXISTS n₂ r] ...] with
[AND ... [EXISTS n r] ...] where $n = \max(n_1, n_2)$

- an expression containing *Thing* with one without *Thing*, unless it is the entire expression. Replace

[AND ... X Y *Thing*...[ALL r *Thing*]] with
[AND ... X Y ...]

- remove any duplicates within an AND expression. Consider another example of a named concept.

```
BComicFilm ≡ [AND Film
                [EXISTS 1 :Villain]
                [EXISTS 2 :Song]
                [EXISTS 1 :Comedian]
                [ALL :Actor [FILLS :Expression funny]]]
```

Now given the two concepts *BComicFilm* and *BFormulaFilm*, we can define a new concept, and its normalized form as follows.

```

BComicFormulaFilm  $\doteq$  [AND BComicFilm BFormulaFilm [FILLS :Producer bcd]]
 $\doteq$  [AND
  [AND Film
    [EXISTS 1 :Villain]
    [EXISTS 2 :Song]
    [EXISTS 1 :Comedian]
    [ALL :Actor [FILLS :Expression funny]]]
  [AND Film
    [EXISTS 2 :Villain]
    [EXISTS 4 :Song]
    [ALL :Actor [AND [FILLS :MinAge 12]
                     [FILLS :LivesIn Mumbai]]]
    [FILLS :Producer bcd]]]
 $\doteq$  [AND Film
  [EXISTS 1 :Villain]
  [EXISTS 2 :Song]
  [EXISTS 1 :Comedian]
  [ALL :Actor [FILLS :Expression funny]]
  Film
  [EXISTS 2 :Villain]
  [EXISTS 4 :Song]
  [ALL :Actor [AND [FILLS :MinAge 12]
                   [FILLS :LivesIn Mumbai]]]
  [FILLS :Producer bcd]]]
 $\doteq$  [AND Film Film
  [EXISTS 1 :Villain] [EXISTS 2 :Villain]
  [EXISTS 2 :Song] [EXISTS 4 :Song]
  [EXISTS 1 :Comedian]
  [ALL :Actor [FILLS :Expression funny]]
  [ALL :Actor [AND [FILLS :MinAge 12]
                   [FILLS :LivesIn Mumbai]]]
  [FILLS :Producer bcd]]]
 $\doteq$  [AND Film
  [EXISTS 2 :Villain]
  [EXISTS 4 :Song]
  [EXISTS 1 :Comedian]
  [ALL :Actor [AND [FILLS :Expression funny]
                   [FILLS :MinAge 12]
                   [FILLS :LivesIn Mumbai]]]
  [FILLS :Producer bcd]]]

```

14.7.4 Structure Matching

Given two descriptions in canonical/normalized form, one can compare the constituents piecewise to determine whether one of them subsumes the other. Let the two descriptions in normalized form be,

$$d_{sub} \doteq [\text{AND } d_1 d_2 \dots d_n] \text{ and } d_{super} \doteq [\text{AND } p_1 p_2 \dots p_k]$$

The interpretation of the AND operator is that it denotes the intersection of the sets denoted by its arguments. The concept d_{sub} is subsumed by the concept d_{super} when the corresponding set $I(d_{sub})$ is a subset of the set $I(d_{super})$. Now if *each* component p_i of d_{super} has a corresponding component d_i in d_{sub} such that $I(d_i) \subseteq I(p_i)$, then the

intersection of all these subsets will be a subset of the intersection of the components of d_{super} , that is d_{super} itself.

Let every component p_i of d_{super} have a corresponding component $d_{i\bar{c}}$ in d_{sub} . Note that it is possible that $d_{k'} = d_{j'}$, that is the same component may be subsumed by different components of d_{super} . Let $I(d_{subsumed})$ be the intersection of all the sets $I(d_1)$, $I(d_2)$, ..., $I(d_n)$. That is, $d_{subsumed}$ is $[AND\ d_1\ d_2\ \dots\ d_n]$. Clearly then,

$$I(d_{subsumed}) \subseteq I(d_{super})$$

Further, since d_{sub} may have more components not included in $d_{subsumed}$,

$$I(d_{sub}) \subseteq I(d_{subsumed})$$

Hence,

$$I(d_{sub}) \subseteq I(d_{super})$$

The relation between these subsets is illustrated in Figure 14.25.

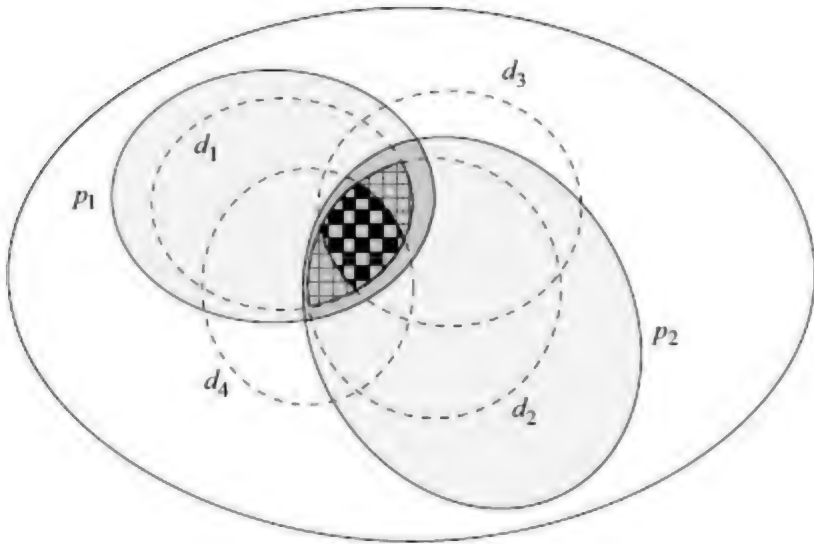


FIGURE 14.25 Piecewise subsumption. Component p_1 subsumes d_1 , and component p_2 subsumes d_2 . The concept $d_{super} \doteq [AND\ p_1\ p_2]$ is the shaded region. The concept $d_{sub} \doteq [AND\ d_1\ d_2\ d_3\ d_4]$ is the region with the chessboard pattern. The concept $d_{subsumed} \doteq [AND\ d_1\ d_2]$ is the region with the grid pattern.

When the task is to show that $(d_{sub} \subseteq d_{super})$, the *structure matching* algorithm looks for a corresponding subsumed element $d_{i'}$ in d_{sub} for every component p_i in d_{super} . The subsumed elements for each type of concept are as follows.

- If p_i is an atomic concept, then it is necessary that $d_{i'} = p_i$

- If $p_i \doteq [\text{EXISTS } n \ r]$, then it is necessary that $d_i \doteq [\text{EXISTS } m \ r]$, where $m \geq n$. In the special case where $n = 1$, it is also allowed that $d_i \doteq [\text{FILLS } r \ c]$ for any constant c
- If $p_i \doteq [\text{ALL } r \ e]$ then it is necessary that $d_i \doteq [\text{ALL } r \ e']$ where recursively $e' \sqsubseteq e$
- If $p_i \doteq [\text{FILLS } r \ c]$ then $d_i \doteq [\text{FILLS } r \ c]$

If the algorithm can find corresponding elements then it returns $(d_{sub} \sqsubseteq d_{super})$.

Observe that we have used $d_i \doteq [\text{FILLS } r \ c]$ to identify a subsumed component for the special case $p_i \doteq [\text{EXISTS } 1 \ r]$. This is because we know that when $[\text{FILLS } r \ c]$ holds then we are talking about some element that is related by r to c , and therefore related at least to one element by r , which is what $p_i \doteq [\text{EXISTS } 1 \ r]$ asserts.

14.7.5 Building Taxonomies

The key difference between Frames and DL based systems is that subsumption is decided by the descriptions in DL systems, while it is encoded in an ad-hoc manner in Frame systems. Having decided upon the basis of subsumption, one can build an explicit taxonomy in which subconcepts are linked to super concepts by links, which stand for the *is-A* relation. The taxonomy can be built automatically by a process known as *classification*, in which one adds new concepts one by one to the taxonomy, and an algorithm places them in the appropriate place.

Let us assume that each concept description d_i in the taxonomy has a name a_i associated with it. That is,

$$a_i \doteq d_i$$

The nodes of the taxonomy are labelled with these atomic concepts (names). Given a taxonomy T and a (new) concept description d , the set S of the *most specific subsumers* of d is the set of all atomic concepts a that subsume d and there are no concepts a' that subsume d and are subsumed by a .

$$S = \{a \mid d \sqsubseteq a \text{ and there is no } a' \text{ s.t. } d \sqsubseteq a' \text{ and } a' \sqsubseteq a\}$$

Likewise, we define the set G of *most specific subsumees* of d as,

$$G = \{a \mid a \sqsubseteq d \text{ and there is no } a' \text{ s.t. } a' \sqsubseteq d \text{ and } a \sqsubseteq a'\}$$

Given a taxonomy T and a *new* concept a_{new} with a description d_{new} , it is placed in the taxonomy T by the following procedure adapted from (Brachman and Levesque, 2004).

```

Classify(taxonomy, aNew)
1  $d \leftarrow \text{Desc}(aNew)$ 
2  $s \leftarrow \text{MostSpecificSubsumers}(taxonomy, d)$ 
3  $g \leftarrow \text{MostGeneralSubsumees}(taxonomy, s, d)$ 
4 if empty( $s \cap g$ )
5   do remove every link from  $g$  to  $s$ 
6     for every  $n \in g$ 
7       add an Is-A link in taxonomy from  $n$  to  $aNew$ 
8     for every  $n \in s$ 
9       add an Is-A link in taxonomy from  $aNew$  to  $s$ 
10 return taxonomy

```

FIGURE 14.26 The procedure *Classify* to insert a node into the taxonomy. We assume a function *Desc(node)* that returns the description of a node. The functions *MostSpecificSubsumers* and *MostGeneralSubsumees* are described in Figures 14.27 and 14.29.

After the sets G and S have been found, if they have a node in common that node already describes the input a_{New} , and nothing needs to be done. Otherwise, we first remove any existing *Is-A* links between G and S (line 4). This is because we want to link nodes only to their parents. Then for every node in G , we add an *Is-A* link to a_{New} , and a link from a_{New} to every node in S .

The set S of most specific subsumers of a concept d in a taxonomy T is constructed as follows. We start at the root, the node for *Thing* that subsumes everything, and traverse downwards along its descendants that subsume the concept description d . We assume a function *Desc(node)* that returns the description associated with a node, and a function *Subsumes(N,M)* that returns true if $M \subseteq N$. The algorithm works with two sets. One called P that contains possible ancestors of the most specific subsumers. The other set S is the set of most specific subsumers we are constructing. For every node in the set P that subsumes d , the algorithm transfers it to S , only if it has no child that subsumes d . Otherwise it replaces it with all its children in P ²³. The algorithm is given in Figure 14.27.

```

MostSpecificSubsumers(taxonomy, d)
1  $p \leftarrow \{\text{root}(taxonomy)\}$ 
2  $s \leftarrow \{\}$ 
3 while not empty( $p$ )
4   do for every node  $n$  in  $p$ 
5     Remove  $n$  from  $p$ 
6     if Subsumes( $\text{Desc}(n)$ ,  $d$ )
7       then if there exists a child  $c$  of  $n$  such that
                                     Subsumes( $\text{Desc}(c)$ ,  $d$ )
8         then Add all children of  $n$  to  $p$ 
9         else Add  $n$  to  $s$ 
10 return  $s$ 

```

FIGURE 14.27 The procedure to find the most specific subsumers of a given description d in a taxonomy. We assume a function *Desc(node)* that returns the description of a node, and a function *Subsumes* that implements structure matching on two descriptions.

The algorithm only traverses that part of the taxonomy downward that

contains nodes that subsume the input concept d . Figure 14.28 illustrates the subgraph explored by the algorithm. The nodes seen by the algorithm are coloured black and white. The black nodes subsume the input concept represented by the shaded square, while the white ones do not, and are discarded by the algorithm. The grey nodes are not inspected by the algorithm. The most specific subsumers are the “lowest” black nodes, and form the set S , shown in the shaded rectangle, that is returned by the algorithm.

The search for the most general subsumees begins with the set S that has been constructed by the algorithm *MostSpecificSubsumers*. The algorithm *MostGeneralSubsumees* traverses the descendants of nodes in S looking for nodes subsumed by the concept d . The algorithm is given in Figure 14.29. We assume that the algorithm is invoked in a call-by-value mode, so that the original set S is not affected.

Figure 14.30 depicts the nodes inspected by the algorithm, coloured black and white. Of these, the black ones are the ones subsumed by d and are added to G , and the white nodes are not and are discarded. The traversal ends in each branch when the algorithm finds a black (subsumed) node, or if there are no children. After the traversal ends, it is possible that there are pairs of nodes X and Y in G such that $Y \subseteq X$. For every such pair, node Y is removed from G before returning G .

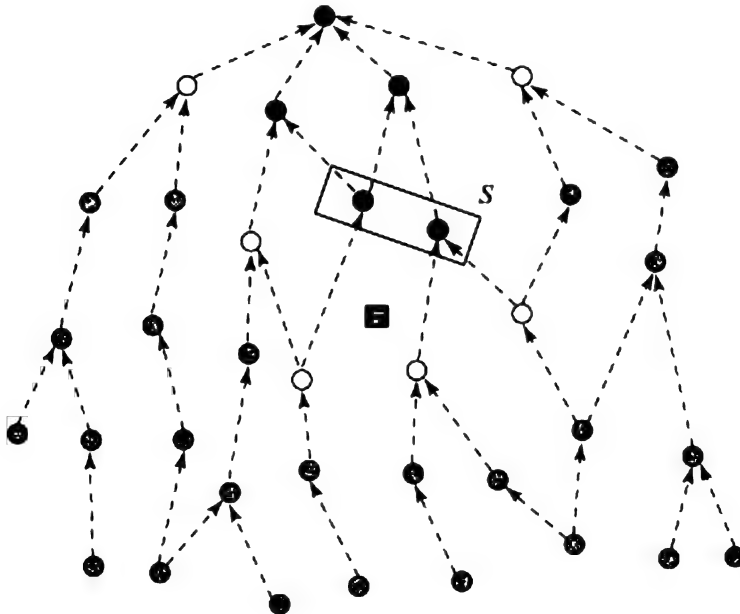


FIGURE 14.28 The nodes of a taxonomy explored by the algorithm *MostSpecificSubsumers*. The black nodes are the nodes that subsume the concept d represented by the square. Of these, the most specific ones are shown by the enclosing shaded rectangle. The white nodes are the ones inspected by the algorithm and discarded.


```

MostGeneralSubsumees(taxonomy, s, d)
1   $g \leftarrow \{ \}$ 
2  while not empty(s)
3    do for every node  $n$  in  $s$ 
4      Remove  $n$  from  $s$ 
5      if Subsumes( $d$ , Desc( $n$ ))
6        then Add  $n$  to  $g$ 
7      else Add all children of  $n$  in taxonomy to  $s$ 
8  if there exist two nodes  $x$  and  $y$  in  $s$  such that Subsumes
                                     (Desc( $x$ ), Desc( $y$ ))
9    then Remove  $y$  from  $g$ 
10 return  $g$ 

```

FIGURE 14.29 The procedure to find the most specific subsumees of a given description d in a taxonomy. We assume a function $Desc(node)$ that returns the description of a node, and a function $Subsumes$ that implements structure matching on two descriptions.

The sets S and G are the nodes that have to be connected to the new concept d_{new} represented by the node a_{new} . The first task, as shown in the algorithm *Classify* described above, is to break all links between nodes in S and nodes in G . After that, *Is-A* links are established between a_{new} and all the nodes in S and in G , as shown in Figure 14.31.

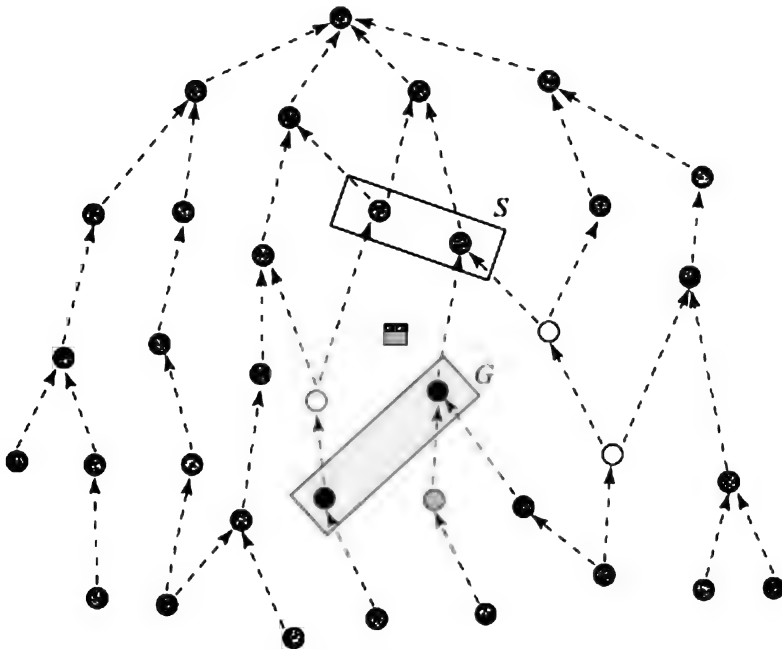


FIGURE 14.30 The algorithm *MostGeneralSubsumees* begins by exploring the forest below the set S , represented by the black and white nodes. The black nodes are the ones subsumed by d , and the set G is the set of "highest" black nodes, as shown.

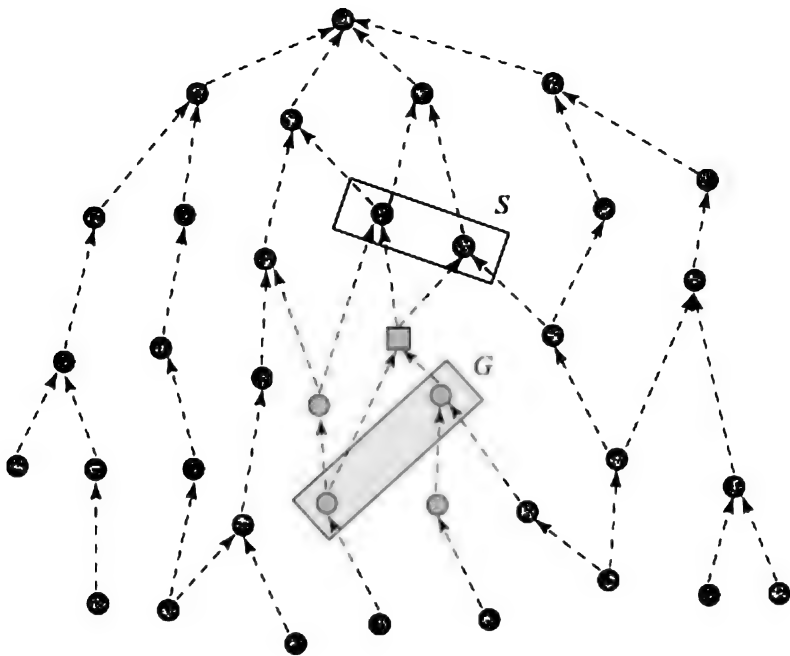


FIGURE 14.31 The new node is inserted into the taxonomy by linking it to nodes in S and in G , after deleting any links between the sets S and G .

Now given a query node q , if we want to find all the nodes that subsume q , we simply have to traverse the taxonomy T upwards from the node q looking for its ancestors. Likewise, to find all concepts subsumed by q , we need to traverse all the descendants of q in T .

14.7.6 Satisfaction

Every constant c in the knowledge base is attached to most specific node a_c in T that it satisfies. If the input assertion is $(c \rightarrow d)$ and a node with description d is not in the taxonomy then a_c is the node that is the most specific subsumer²⁴ of the description d .

Then, if there is a query asking whether c satisfies a concept q , we can translate it to a query in which we ask if d_c is subsumed by q , where d_c is the description of a_c , that the individual c is attached to.

Observe that the input sentence $(c \rightarrow d)$ has to be constructed explicitly from a database. Assuming that the database lists all roles for an individually and their fillers explicitly, the descriptions $[\text{ALL } r \ d]$ and $[\text{EXISTS } n \ r]$ can be made about an individual only by inspecting the database and making the relevant inferences.

However, we have to be careful in choosing the input assertion $(c \rightarrow d)$ while inserting the constant in T . The concept description d in $(c \rightarrow d)$ must contain *everything* that we know about the constant c . Sometimes this

information is not explicit in the set of sentences of DL available, and may have to be ferreted out by a process of making inferences. Consider the following example,

jaaneBTY \rightarrow [AND BComicFilm [FILLS :Actor sShah]]
sShah \rightarrow [AND Indian [EXISTS 10000000 :Fan]]

The first statement says that *jaaneBTY* is a *BComicFilm* in which *sShah* has acted. The second one tells us that *sShah* is an Indian with more than ten million fans. But consider what we already know about *BComicFilms* reproduced again below.

BComicFilm \doteq [AND Film	
	[EXISTS 1 :Villain]
	[EXISTS 2 :Song]
	[EXISTS 1 :Comedian]
	[ALL :Actor [FILLS :Expression funny]]]

If the above three sentences are our knowledge base, then what do we know about the actor *sShah*? We know that he is an Indian with more than ten million fans because that has been stated explicitly. But we also know that *BComicFilms* are those all whose actors have funny expressions. So this must apply to *sShah* as well, because he acted in *jaaneBTY* which is a *BComicFilm*. So when we insert the constant *sShah* into the taxonomy, we must do so with the sentence,

sShah \rightarrow [AND Indian [EXISTS 10000000 :Fan] [FILLS :Expression funny]]

That is, we first need to collect all information about *sShah* that is entailed by the knowledge base before deciding where to attach the constant.

We look at another example. Let Abheek be a person who is only a fan of football players, and also that he is a fan of Gerrard. Suppose all we know about Gerrard is that he is an Englishman,

abheek \rightarrow [AND Person [FILLS :FanOf gerrard] [ALL :FanOf Footballer]]
gerrard \rightarrow [AND Person English]

Then we should augment our information of Gerrard to,

gerrard \rightarrow [AND Person English Footballer]

In the forward-chaining algorithm *PropagateProperties* that follows in Figure 14.32, the following correspondence holds,

$c_1 \leftrightarrow abheek$
 $d_1 \leftrightarrow$ [AND ... [FILLS :FanOf gerrard] [ALL :FanOf Footballer]]
 $c_2 \leftrightarrow gerrard$

$d_2 \leftrightarrow [\text{AND Person English}]$

```

PropagateProperties(kb)
1 for every constant c in the kb
2 form the normalized sentence (c→d) by combining all such sentences
3 while there exists c1 and c2 s.t. C2inC1(c1→d1, c2→d2, kb) ≠ nil
4 do e ← C2inC1(c1, c2)
5 d2 ← [AND d2 e]
6 Normalize d2
7 Update (c2→d2) in the kb
8 return kb

C2inC1(c1→d1, c2→d2, kb)
1 if [FILLS r c2] and [ALL r e] are components of d1
2 then return e
3 else return nil

```

FIGURE 14.32 The procedure to augment the descriptions of all constants. For every pair of descriptions of two constants, if one contains implicit information about the other, the other description is augmented.

The above algorithm is a forward-chaining algorithm that adds any inferences that can be made about an individual to the knowledge base. The inference is made based on the occurrences of two descriptions that apply to the same constant c_1 . The descriptions are $[\text{FILLS } r \ c_2]$ and $[\text{ALL } r \ e]$, and contain implicitly the information that c_2 must belong to the class e as well. However, this pair of descriptions could have been buried deeper in some description.

In the example described above, we had some descriptions for Abheek. But we could have had these descriptions in an indirect manner as shown below.

```

aasutosh → [AND [EXISTS 1 :Friend]
               [ALL :Friend [AND Person [FILLS :FanOf gerrard]
                                         [ALL :FanOf Footballer]]]]
gerrard → [AND Person English]

```

The first sentence is now about a different person Aasutosh, and what we know about him is that he has at least one friend, and that all his friends are fans of Gerrard and are fans only of footballers. Abheek could have been one of these friends, but has not been named. In fact, none of the friends has been named. Yet these two sentences contain the same extra information about Gerrard.

One can get around this problem by keeping data about anonymous constants just like named constants. The anonymous individuals we have to deal with are essentially related by some role, or sequence of roles to some known individual. We represent them by adding role names to the known individual for example, *aasutosh.Friend*. Of course, there may be a longer connection like for example *abheek.Father.Friend.FanOf* which would stand for the person who Abheek's father's friend is a fan of. In general, we denote such role chains by letters like σ and δ .

In the example containing Abheek, the two individuals involved in the

rule of inference were *abheek* and *abheek.FanOf*. The latter refers to any person who Abheek is a fan of, but becomes identified with Gerrard because of the description [FILLS :FanOf gerrard] in Abheek's description. In the case of anonymous individuals like the friends of Aasutosh, the corresponding elements are *aasutosh.Friend* and *aasutosh.Friend.FanOf*. The inference rule may be generalized as follows,

If there exists a constant c_1 and a (possibly empty) role chain σ such that we can add $(c_1 \cdot \sigma, d_1)$ to the knowledge base, and that we can add $(c_1 \cdot \sigma \cdot r, d_2)$, and $c_1 \cdot \sigma \cdot r$ is identified with c_2 by means of a [FILLS $r c_2$] statement, and if d_1 contains the information [EXISTS 1 r] and [ALL $r e$] then we can augment the information about c_2 to the normalized version of [AND $d_2 e$].

14.7.7 Limitations of DL

Description logics give us a logical basis for constructing taxonomies. We are compelled to be precise, and the super class subclass relations are determined strictly by the descriptions. The fact that we need precise descriptions will not allow koalas to be described as a type of bear, or whales to be a type of large fish, or a civet to be a type of cat. Inheritance, or subsumption, is strict in DL. If a concept d_{sub} is subsumed by a concept d_{super} then elements of d_{sub} must inherit properties associated with d_{super} . There is no question of exceptions.

The class-subclass inference in DL is determined by structure matching which has time complexity linear in the length of the descriptions, as opposed to theorem proving in FOL that can, in the worst case, be intractable. Observe that though there is a recursive element in matching components of the form [ALL $r d$], the algorithm works with normalized expressions which have the recursive definitions made explicit, resulting in correspondingly longer expressions.

However, the structure matching algorithm works only for a subset of concepts that can be defined in FOL. Thus, we trade expressivity for computational complexity when we choose to represent concepts in Description Logic.

The non-logical elements of the language are of three kinds: concept names, relation names, and constants. The concept forming operators we have used are AND, EXISTS, ALL, and FILLS.

The operator [EXISTS $n r$] says that the elements of the set (concept) must be related by r to at least n elements. We can define another operator [AT-MOST $n r$] that says the elements must be connected to at most n elements (Brachman and Levesque, 2004). It turns out that if we do that then structure matching would no longer work.

Consider the class of families, all whose members are wizards and in which there is exactly one child named Harry.

$Pfamily \doteq [AND [ALL :Member Wiz] [FILLS :Member harry] [EXISTS 1 :Child] [AT-MOST 1 :Child] [FILLS :Child harry]]$

This class would include the Potter family,

$pFamily \rightarrow [AND Pfamily [FILLS :Member james] [FILLS :Member lily]]$

Now the *Pfamily* concept is a subclass of the *Wfamily* concept in which *all* the kids are wizards,

$Wfamily \doteq [ALL :Child Wiz]$

and

$Pfamily \sqsubseteq Wfamily$

But it is clear that the structure matching is unable to conclude that the *Pfamily* is subsumed by the *Wfamily* (because the *Pfamily* simply does not have a clause of the type $[ALL :Child \dots]$). The subsumption holds because Harry, the *only* child in the *Pfamily* class, is a wizard because all members of *Pfamily* are wizards stated in $[ALL :Member Wiz]$. Hence, *Pfamily* is a type of family in which all the children are wizards. Structure mapping cannot cater to such interactions between different clauses in a description.

New concepts are defined by the AND operator as a combination of constituent concepts. However, as observed above, DL cannot cater to interaction between the concepts. We consider another problem of *defining* the set of aunts or the concept *Aunt*. A woman is an aunt if she has a sibling who has a child. Or, a woman is an aunt if she has a sibling who is a parent. The fact that she must have a sibling can be handled by EXISTS operator with the *:Sibling* role. But the constraint that the sibling must be a parent cannot be handled. One can define the set of parents, and the set of people with siblings as,

$Parent \doteq [EXISTS 1 :Child]$
 $HasSibling \doteq [EXISTS 1 :Sibling]$

But we are unable to define the concept of aunt as *someone who has a sibling who has a child*. We could extend our DL by one of the following operators,

$[D-FILLS \ r \ d]$: like FILLS, but the second argument is a concept

or

$[D-EXISTS \ n \ r \ d]$: related by role *r* to *n* elements of type *d*

Observe that both are similar and existential in nature, saying that there exists a filler that belongs to a certain concept; or we could

introduce a subsumption hierarchy in roles and create a new role by adding an argument to role that restricts the type of elements it can relate to,

[RESTR $r\ d$] : a specialization of role r that relates to elements of type d

For example the role *:Mother* is the same as [RESTR :Parent Female]. We could then define aunts with each of the three new operators, as follows,

Aunt \doteq [AND Person Female [EXISTS 1 :Sibling] [D-FILLS :Sibling Parent]]

or

Aunt \doteq [AND Person Female [D-EXISTS 1 :Sibling Parent]]

or

Aunt \doteq [AND Person Female [EXISTS 1 [RESTR :Sibling Parent]]]

However, while these extensions allow us to define the concept of aunt, it is no longer possible to use structure matching to decide whether one concept is subsumed by another. In the following definition, we assume that [EXISTS $n\ r$] is a short form for [D-EXISTS $n\ r\ Thing$] to allow possible matching. Consider a woman who is related to at least two siblings and all whose siblings are parents. She could be defined in our original DL as follows,

Aunty \doteq [AND Person Female [EXISTS 2 :Sibling] [ALL :Sibling Parent]]

Clearly, the concept *Aunty* is subsumed by concept *Aunt*, because an individual who satisfies *Aunty* also satisfies *Aunt*. But we are unable to compare the components in description of *Aunty* with any of the components in the description of *Aunt* in a piecewise fashion required by structure matching. And structure matching is the procedure that allows us to answer the question of subsumption by just inspecting the two descriptions. Theorem proving in FOL also allows us to compute subsumption, but with no guarantees of speed.

Given a domain, DL is a language that talks about subsets in the domain. It allows one level of abstraction in Figure 14.1. It does not allow us to talk about aggregation resulting in other entities extended to reified elements. Extending the DL beyond its strict boundaries can only be done at the cost of foregoing the computationally cheaper structure-matching process.

Box 14.2: A Family of Description Logics

Description Logics is in fact a name for a family of logic-based languages to represent knowledge of domains (Baader and Nutt, 2003). Each language is a little different in expressiveness and consequently has different complexity of making inferences. What is common amongst all is that the basic building blocks in all are concepts, roles, and individuals. Traditionally, the knowledge base is partitioned into two parts. The *TBox* is concerned with terminological reasoning, or reasoning about concepts. The *ABox* is concerned with assertions, or reasoning about individuals.

A simple language named *AL* for *attributive languages* was introduced by Schmidt-Schauß and Smolka (1991). The *AL* allows the following expressions for describing concepts.

- A or \perp or \top are atomic concepts
- $\neg A$ is the negation of an atomic concept
- $C \sqcap D$ is the intersection of concepts C and D (equivalent to $[\text{AND } C \ D]$)
- $\forall R.C$ restricts all role fillers to C (equivalent to $[\text{ALL } :R \ C]$)
- $\exists R.\top$ says that there exists a role filler (equivalent to $[\text{EXISTS } 1 :R]$)

An even simpler language traditionally known as FL^- is a sublanguage of *AL* without negation. If we further disallow the existential statement, the language is known as FL_0 . The more recent nomenclature described below extends *AL*.

More expressive languages can be obtained by adding some of the following descriptors.

- $C \sqcup D$ is the union of concepts C and D . The corresponding language is described using the additional symbol \cup , for example $\text{AL}\cup$.
- $\exists R.C$ says that there exists a role filler of type C . The language is characterized by the (additional) symbol \exists , for full existential quantification.
- $\geq nR$ says that there are at least n role fillers, and $\leq nR$ says that there are at most n role fillers. The extended language is characterized by \mathbb{N} , for number restriction.
- $\leq 1R$ is a functional relation characterized by F .
- $\geq nR.C$ and $\leq nR.C$ are qualified number restrictions in which the role filler belongs to the concept C . These languages are marked with Q .
- $\neg C$ is the negation of arbitrary concepts, allowed in languages characterized by C .
- $\{a_1, \dots, a_n\}$ for $n \geq 1$ allows concepts to be described explicitly by listing the individuals. The language extension is O .

Thus, extensions may be described as $\text{AL}\cup\exists\mathbb{N}\text{FQO}$, depending upon the additional descriptors.

In addition to allowing different concept forming operators, description languages may also allow one to specify roles in terms of other roles. Thus, we may have role inverse, role intersection, role union, role complement and role composition operators to form new roles.

Description languages may be characterized by role transitivity (characterized by S) role hierarchy (like the RESTR operator described earlier, characterized by H), and complex role inclusion (characterized by R). See also Section 14.7.8 on OWL.

The interested reader is encouraged to visit the site (Zolin) on DL complexity maintained by Evgeny Zolin of Manchester University, UK.

14.7.8 The Web Ontology Language

The most widely accepted definition of an ontology is due to Tom Gruber —“*an ontology is a specification of a conceptualization*” (Gruber, 1993). The Free Online Dictionary of Computing expands the definition²⁵—“*An explicit formal specification of how to represent the objects, concepts and other entities that are assumed to exist in some area of interest and the relationships that hold among them. ... For AI systems, what “exists” is that which can be represented.*” The ontology provides a framework for knowledge representation.

In a broader sense, an ontology is a study of things that *might* exist. From the philosophical point of view “*Ontology seeks to provide a definitive and exhaustive classification of entities in all spheres of being*” (Smith, 2003). An ontology specifies the classes of entities that might be needed to completely specify knowledge in a domain²⁶. Interest in Ontology (or ontologies as we often say) increased manifold with the emergence of the World Wide Web. The Web is not just a medium for people accessing pages put up by others, but also for programs accessing data hosted in remote systems and sending data to other systems. The ontology associated with a system becomes the basis of defining the semantics of the terms used. One of the driving forces is the business of e-commerce in which for example an internet agent can visit many sites looking for best prices for products with a given description.

The WWW consortium has converged on a language for specifying ontology on the Web. This language is called OWL, and stands for Web Ontology Language²⁷. OWL builds upon RDF and RDF Schema (see Chapter 13). The most commonly used syntax is based on XML, though it is not an integral part of OWL.

OWL is used to define resources and properties about resources. The language comes in three flavours.

OWL Full The full language allows complete expressivity. OWL Full allows reification. Classes can be treated as collections of objects from

the domain, as well as objects themselves. Everything that can be expressed in FOL can be expressed in OWL Full. As a corollary, reasoning in OWL Full can be undecidable in the worst case.

OWL DL A subset of OWL Full that conforms to Description Logic. It prohibits interaction between different components in the representation and consequently can allow computationally efficient reasoning.

OWL Lite A further restriction on the language that restricts it mostly to the class hierarchy with simple constraints on features.

We look at the full language briefly. OWL is concerned with defining concepts as class descriptions and their properties or roles. The *owl:Class* is an *rdfs:Class* which is an *rdfs:Resource*. The properties in OWL are of two types. The first, *owl:ObjectProperty*, is a role that takes another concept or class as a filler. The *owl:ObjectProperty* specifies a binary relation between two objects in the domain. The second, *owl:DatatypeProperty*, relates an *owl:Class* element to values from a data type. Each of the three, the *owl:Class*, *owl:ObjectProperty*, and *owl:DatatypeProperty* is an *rdfs:Resource*.

Every OWL document begins with a header specifying a number of namespaces.

```
<rdf:RDF
  xmlns:owl = "http://www.w3.org/2002/07/owl#"
  xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs = "http://www.w3.org/2000/01/rdf-schema#"
  xmlns:xsd = "http://www.w3.org/2001/XMLSchema#">
```

OWL definitions usually give names to classes and properties, though anonymous definitions are possible. An OWL class may be defined, for example, as follows.

```
<owl:Class rdf:ID = "khayal">
  <rdfs:subClassOf ref:resource = "#hindustani"/>
</owl:Class>
or
<owl:Class rdf:ID = "vocalist">
  <rdfs:subClassOf ref:resource = "#musician"/>
</owl:Class>
```

We can relate a named class to other classes by asserting that they are *equivalent* or that they are *disjoint*,

```
<owl:Class rdf:about = "#hindustani">
  <owl:equivalentClass rdf:resource = "#northIndianClassical"/>
  <owl:disjointWith rdf:resource = "#carnatic"/>
  <owl:disjointWith rdf:resource = "#jazz"/>
  <owl:disjointWith rdf:resource = "#indiPop"/>
  <owl:disjointWith rdf:resource = "#westernClassical"/>
</owl:Class>
```

The first line in the above uses “#hindustani” to refer to the class *hindustani*, defined elsewhere. We can define a class to be the *complement*, *union* or *intersection* of other, possibly anonymous classes.

```
<owl:Class rdf:ID="indianMusic">
  <owl:unionOf rdf:parseType = "Collection">
    <owl:Class rdf:about = "#hindustani"/>
    <owl:Class rdf:about = "#carnatic"/>
    <owl:Class rdf:about = "#folkMusic"/>
    <owl:Class rdf:about = "#filmMusic"/>
  </owl:unionOf>
</owl:Class>
and,
<owl:Class rdf:ID = "bhimsenKhayals">
  <owl:intersectionOf rdf:parseType = "Collection">
    <owl:Class rdf:about = "#khayal"/>
    <owl:Restriction>
      <owl:onProperty rdf:resource = "singer"/>
      <owl:hasValue rdf:resource = "#bhimsenJoshi"/>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

The last example above states that the set of khayals sung by Bhimsen Joshi is the intersection of the named set *khayal* and the anonymous set which has a property restriction that limits the set of compositions to the ones where the artist is Bhimsen Joshi. The following definition assumes that the concepts *human* and *wizard* have been defined already. It says that *muggles* are those humans that are not wizards.

```
<owl:Class rdf:ID = "muggle">
  <owl:intersectionOf rdf:parseType = "Collection">
    <owl:Class rdf:about = "#human"/>
    <owl:Restriction>
      <owl:complementOf rdf:resource = "#wizard"/>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

New concepts may also be defined by *constraints* on the property fillers. The following definition of a data type property called *year* says that *year* is a value of type *nonNegativeInteger*, defined in the XML namespace.

```
<owl:DatatypeProperty rdf:ID = "year">
  <rdfs:range rdf:resource = "http://www.w3.org/2001/XMLSchema
    #nonNegativeInteger"/>
</owl:DatatypeProperty>
```

Object properties on the other hand relate objects of a particular class

to objects of another class. The following example relates a music composition to the composer. It also says that the *composedBy* property is a subproperty of *musician*, is the inverse of property *composition* (that would relate a composer to a composition), and is also known by the name *musicScoreBy*. Observe that OWL allows one to organize properties into a hierarchy, and allows one property to be declared as the inverse of another, or equivalent to another.

```
<owl:ObjectProperty rdf:ID = "composedBy">
  <owl:domain rdf:resource = "#musicNumber"/>
  <owl:range rdf:resource = "#composer"/>
  <rdfs:subPropertyOf rdf:resource = "#musician"/>
  <owl:inverseOf rdf:resource = "#composition"/>
  <owl:equivalentProperty rdf:resource = "#musicScoreBy"/>
</owl:ObjectProperty>
```

The following constraints on properties, specified by *owl:Restriction*, are equivalent to [ALL *r d*], [FILLS *r c*], and [EXISTS 1 *r*] of DL respectively.

Let us say we define a class called *9symphonies* that contains the nine symphonies composed by Beethoven. We could then define the class of people *beenutShroeder* who are real fans of Beethoven's symphonies by using the *owl:allValuesFrom*, which is the universally quantified statement of DL. The following statement says that *beenutShroeder* is the set of people who love to listen *only to* Beethoven's symphonies.

```
<owl:Class rdf:ID = "beenutShroeder">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource = "#lovesToListen"/>
      <owl:allValuesFrom rdf:resource = "#9symphonies"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

In our DL, we would have expressed this as [ALL :lovesToListen 9symphonies]. We can define the set (or class) *9symphonies* by specifying the composer, in the manner of [FILLS *r c*] of DL.

```
<owl:Class rdf:ID = "9symphonies">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource = "#composedBy"/>
      <owl:hasValue rdf:resource = "#ludwigVanBeethoven"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

We can define the set of people *realMusicLover* who like to listen to at least one of Beethoven's symphonies, whatever else they may listen to. This is equivalent to the extension [D-EXISTS *n r d*] defined in the

previous section, with $n = 1$. Equivalently, we might think of the statement below as $[\text{EXISTS } 1 \text{ } r']$ where $r' = [\text{RESTR } r \text{ } d]$.

```
<owl:Class rdf:ID = "realMusicLover">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource = "# lovesToListen"/>
      <owl:someValuesFrom rdf:resource = "#9symphonies"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Observe that in all the three definitions above, the class is defined as a subclass of an unnamed class defined by the *owl:Restriction* construct.

OWL also allows one to define a class by enumerating its elements using the *owl:oneOf* construct. For example, we can define the set of symphonies by Beethoven as,

```
<owl:Class rdf:ID = "symphoniesOfBeethoven">
  <owl:oneOf rdf:parseType = "Collection">
    <owl:Thing rdf:about = "# No.1, Op.21, C Major"/>
    <owl:Thing rdf:about = "# No.2, Op.36, D Major"/>
    <owl:Thing rdf:about = "# No.3, Op.55, E flat Major : Eroica"/>
    <owl:Thing rdf:about = "# No.4, Op.60, B flat Major"/>
    <owl:Thing rdf:about = "# No.5, Op.67, C Minor"/>
    <owl:Thing rdf:about = "# No.6, Op.68, F Major : Pastoral"/>
    <owl:Thing rdf:about = "# No.7, Op.92, A Major"/>
    <owl:Thing rdf:about = "# No.8, Op.93, F Major"/>
    <owl:Thing rdf:about = "# No. 9, Op. 125, D Minor : Choral"/>
  </owl:oneOf>
</owl:Class>
```

We can also say somewhere that the class *9symphonies* is the same as (using *owl:sameAs*) the class *symphoniesOfBeethoven*.

Instances may be added to a knowledge base using the following OWL statements,

```
<rdf:Description rdf:ID = "leningrad">
  <rdf:type rdf:resource = "#symphony"/>
</rdf:Description>

<symphony rdf:ID = "leningrad">
  <musicSite:composer = "#dmitriShostakovich">
</academicStaffMember>
```

The first statement says that *leningrad* is an element of type *symphony*, defined elsewhere. The second one says that for the element *leningrad*, the property *composer* defined in the *musicSite* namespace has the value object *dimitriShostakovich* defined elsewhere. We can assert that tokens *dimitriShostakovich* and *ludwigVanBeethoven* refer to

different individuals. Note that this has to be done explicitly, since OWL does not make the *unique name assumption* (UNA) that says that each instance has a unique name.

```
<composer rdf:about = "dimitriShostakovich">
  <owl:differentFrom rdf:resource = "ludwigVanBeethoven"/>
</composer>
```

One can say that a collection of tokens refer to distinct individuals.

```
<owl:allDifferent>
  <owl:distinctMembers rdf:parseType = "Collection">
    <composer rdf:about = "dimitriShostakovich "/>
    <composer rdf:about = "ludwigVanBeethoven"/>
    <composer rdf:about = "wolfgangAmadeusMozart"/>
    <composer rdf:about = "johannSebastianBach"/>
  </owl:distinctMembers>
</owl:allDifferent>
```

OWL also allows cardinality restrictions to be placed in the manner of [EXISTS n r] and [AT-MOST n r]. This is expressed by *owl:minCardinality* and *owl:maxCardinality*.

```
<owl:Class rdf:about = "averageFan">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource = "#fanOf"/>
      <owl:maxCardinality rdf:datatype = "&xsd;nonNegativeInteger">
        4
      </owl:maxCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

The above statement says that the class *averageFan* is made up of elements (people) who are fans of at most 4 elements (people).

The thought might have occurred to the discerning reader that OWL requires a considerable amount of overheads in terms of space, and that the XML based syntax is very cumbersome. The response to the second point is that the syntax is meant to be processed by a program and not by a human. Humans will presumably define and build their knowledge bases using some easy to use interface, for example Protégé²⁸ (see also (Denny, 2002)).

The answer to the first question is that OWL is a language for *specifying* ontologies. It need not be a language for storing ontologies and the instances associated with them. Having a formal specification of the ontology will allow one to reason about it. One can ask questions of the type we had posed in the Description Logic section—does an individual belong to a particular class or, does one class subsume another, or is the specification consistent? If ontology will have a large amount of instances or individuals stored with it, then it will be profitable

to pursue computationally efficient storage systems. For example, we might want to store an ontology in a relational data base management system (see for example (Pan and Heflin, 2003), (Kraska and Röhm, 2006), (Rohloff et al., 2007)).

While the ontology language leads to overheads in space requirement, the XML-based tags become useful when software agents across the Web have to communicate. Then the ontology becomes a formal description of the language used by each system, describing the entities and the relations between them. Then a set of agents that share the same ontology will be able to communicate meaningfully amongst themselves, and for other agents, one could rely on a “translator” that maps entities of one ontology to another.

The role of Ontologies has become prominent with the possibility of a semantic web (Berners-Lee et al., 2001). The possibility of programs exchanging information over the web has brought about the need of shared vocabularies and conceptualizations.

14.8 Formal Concept Analysis

A *lattice* is a partial order in which every two elements have a unique *supremum* and an *infimum*. In the taxonomy, a supremum corresponds to the most specific common subsumer of two nodes, and the infimum corresponds to the most general common subsumee.

Formal Concept Analysis (FCA) is a method of automatically deriving a taxonomy of concepts from a collection of *objects* and their *properties* (Wille, 1982), (Ganter et al., 2005). Each concept, as defined below, relates a set of objects to a set of attributes. The concepts formed depend directly on the information about objects and their properties, and can be mined automatically. The concepts also place themselves in a lattice structure.

FCA defines *formal concepts* in the *setting* of a *formal context*. A formal context contains all the (relevant) information about all the *objects* in the domain in terms of binary valued properties. The formal context has a *yes* or *no* answer for every property for every object. The formal context can be simply represented by an incidence matrix for a bipartite graph between objects and properties. Each edge from an object links it to a property that holds for the object.

Let $K = (G, M, R)$ be a formal context defined by,

- G the set of objects (from the German word *Gegenstände*)
- M the set of properties (from *Merkmale*), and
- $R \subseteq G \times M$ is a relation, such that $\langle g, m \rangle \in R$ iff object g has the property m

The following table depicts a context in which objects, labelled A–F, have properties labelled 1–4 indicated by a cross in the relevant place.

Table 14.1 An example context. The objects labelled A–F have properties labelled 1–4, as shown in the table. An “X” says that the object in that row has the property in that column

Objects	Properties			
	1	2	3	4
A	X			X
B		X	X	
C	X	X	X	
D				X
E			X	
F		X		

Given the context K , a set of formal concepts is defined as follows (Krötzsch and Ganter, 2009). Let att and obj be two functions on the power sets of G and M ,

$$att: 2^G \rightarrow 2^M \text{ and } obj: 2^M \rightarrow 2^G$$

For a set of objects $O \subseteq G$, the set O' of corresponding properties is defined as,

$$O' = att(O) = \{m \in M \mid \langle g, m \rangle \in R \text{ for all } g \in O\}$$

The set O' is the set of properties associated with the set of objects O . That is, each element in O has each property in O' . Likewise, for a chosen set of properties P , we can define the corresponding set of objects, all of whom have all the properties.

$$P' = obj(P) = \{g \in G \mid \langle g, m \rangle \in R \text{ for all } m \in P\}$$

The pair $C = (O, P)$ is a formal concept iff,

$$O = P' \text{ and } P = O'$$

Then O is called the *extension* of concept C and P the *intension* of the concept²⁹.

$$O = ext(C) \text{ and } P = int(C)$$

A formal concept is not a category, but represents an understanding of the given data of objects and their properties. Formal concepts are mathematical constructs and not formal, logical entities (Wille, 2005).

A concept is a set of objects which share some properties. Both the sets, objects and properties, have to be maximal in the sense described below. A formal concept captures information both in the form of an *intension* and in the form of an *extension*. As an extension, it specifies *all the objects* that have a set of properties (specified by the intension) in common. Any object that does not belong to the concept does not satisfy some property in the intension. As an intension, the concept specifies *all the properties* that hold for all the objects (specified by the extension) in the concept. Any property that is excluded has some object in the extension that does not satisfy it. Only those objects with their corresponding properties that satisfy the above define a concept.

A set of objects O is said to be *closed* if $O = (O')' = \text{obj}(\text{att}(O))$ written as $O = O''$. Likewise, a set of properties P is said to be closed if $P = P'' = \text{att}(\text{obj}(P))$. A formal concept can also be defined as the pair (O'', P'') .

A simple method to find all the concepts in a context would be to look at all subsets of G and check whether they are closed, and checking whether their corresponding property sets are closed too. But this procedure involves looking at all subsets of G , or the power set of G , which is exponential in size. The set of all concepts of a context $K(G, M, R)$ is denoted by $\mathcal{B}(G, M, R)$.

Given a formal context $K(G, M, R)$ and the induced set of formal concepts $\mathcal{B}(G, M, R)$, the concepts can be structured into a lattice by using the following relation. The formal concept $(O_{\text{sub}}, P_{\text{sub}})$ is a subconcept of a concept $(O_{\text{super}}, P_{\text{super}})$ iff $O_{\text{sub}} \subseteq O_{\text{super}}$ or equivalently $P_{\text{super}} \subseteq P_{\text{sub}}$.

$$(O_{\text{sub}}, P_{\text{sub}}) \prec (O_{\text{super}}, P_{\text{super}}) \equiv O_{\text{sub}} \subseteq O_{\text{super}}$$

or

$$(O_{\text{sub}}, P_{\text{sub}}) \prec (O_{\text{super}}, P_{\text{super}}) \equiv P_{\text{super}} \subseteq P_{\text{sub}}$$

The orderings induced by the subset relations for sets of objects is the opposite of the ordering induced by the subset relation on the sets of attributes in the lattice. This connection between the two is known as the *antitone Galois connection*, after the French mathematician Évariste Galois. Any one of them can uniquely determine the lattice, which is also known as *Treillis de Galois*³⁰.

The lattice for the concepts in the context defined in Table 14.1 is shown in Figure 14.33 below. The diagram is a *Hasse diagram* used to depict partially ordered sets. The Hasse diagram uses undirected edges with the convention that if $C_1 \prec C_2$ and there is no C_m such that $C_1 \prec C_m \prec C_2$ then C_1 is drawn below C_2 . No transitive edges are drawn. When an edge links C_1 and C_2 and $C_1 \prec C_2$ then C_1 is called the *successor* or *lower neighbour* of C_2 and C_2 is called the *predecessor* or *upper neighbour* of C_1 .

$$\text{Successor}(C_1, C_2) = (C_1 \prec C_2) \text{ and there is no } C_K \text{ such that } (C_1 \prec C_m) \text{ and } (C_m \prec C_2)$$

The top-level concept contains the set of all objects depicted conventionally by $ABCDE$ ³¹ and the associated set of properties, which in this example, is the empty set. Likewise, the least concept contains the set of all properties, depicted by 1234, and the associated set of objects, which in this example is the empty set. In the line diagram in Figure 14.33, each formal concept is labelled with both the set of objects and the set of properties.

While the labelling in the above diagram is explicit, and one can read

off the constituents of a concept at each node, it can become cluttered if the context is large. The following observation can lead to a simpler labelling.

If an object occurs in a concept then it also occurs in any concept higher in the lattice. This follows from the definition of the ordering relation between formal concepts. Further, it can be seen that a subgraph of concepts containing an object g has a unique infimum. This node can then be labelled with g with the understanding that all nodes higher in the ordering also contain g . Likewise, one can find a unique highest concept in the lattice that is labelled with a given property m , and this can be labelled with m , and it is understood that all nodes lower in the partial order also have the property m .

The lattice with this succinct labelling is shown in Figure 14.34. The diagram also contains nodes that are fully or partially shaded. The upper half is shaded whenever the node gets labelled with a property, the lower half if it gets labelled with an object, and the node is fully shaded if it has both kinds of labels.

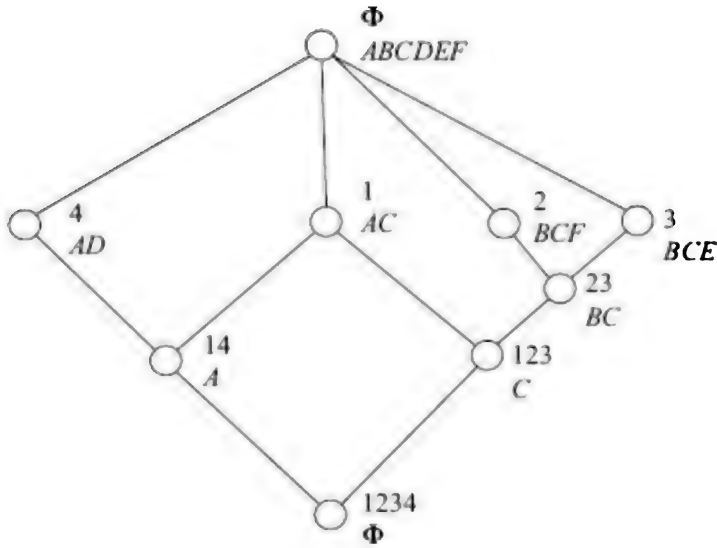


FIGURE 14.33 The concept lattice for the context from Table 14.1. The supremum is the set of all objects, and the infimum, the set of all properties. In this example, the supremum has no common property, and the infimum has no object having all properties.

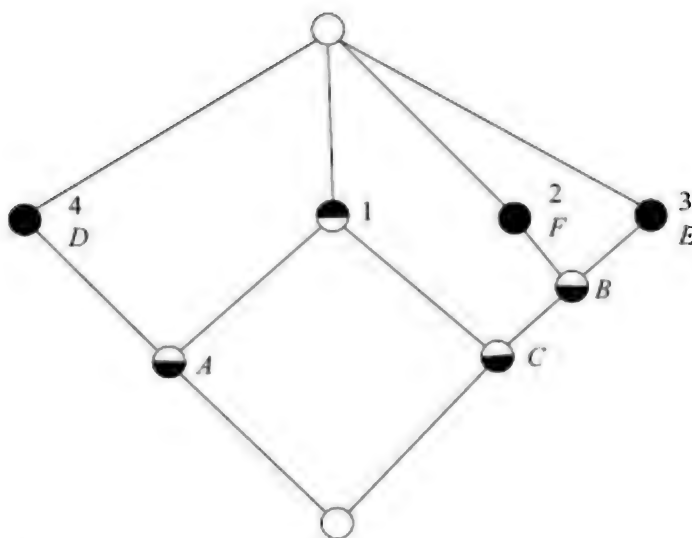


FIGURE 14.34 A simpler labelling in which each object and each property occurs exactly once in the diagram. The labels occur at shaded circles. The upper half is shaded for properties and the lower half for objects.

A concept represents a maximal subarray of X 's in the table that represents the formal context that can be formed with the combination of objects and properties by row and column interchanges.

The following table gives another example of a context in which the objects are students and the properties the subject periods they like. There are four concepts, two of which are marked in the table, one with shaded squares ($\{\text{aarti, amrita}\}, \{\text{science, games}\}$), and the other with a thicker boundary ($\{\text{avinash, arnav, aarti, amrita, ashwani}\}, \{\text{science}\}$). In the example constructed here, no shuffling of rows and columns is needed. It must be noted that all concepts need not appear as contiguous maximal rectangles at the same time, even though it is so in this example.

Table 14.2 Another example context. A concept in a context is a maximal rectangle of X 's that one can obtain after shuffling rows and columns. Two of the four concepts are marked. One with a thick boundary, the other with shaded squares

Objects	Properties					
	math	english	science	games	music	history
avinash			X			
arnav		X	X			
aarti	X	X	X	X	X	
amrita			X	X		
ashwani			X			
ashok						

The concept lattice for the above concept is given in Figure 14.35. The four concepts in the above context are,

- $$C_1 = (\{aarti\}, \{\text{math, english, science, games, music}\})$$
- $$C_2 = (\{\text{arnav, aarti}\}, \{\text{english, science}\})$$
- $$C_3 = (\{aarti, amrita\}, \{\text{science, games}\})$$
- $$C_4 = (\{\text{avinash, arnav, aarti, amrita, ashwani}\}, \{\text{science}\})$$

Apart from this, there is the top concept containing all the students and no subject, and the bottom concept containing all the subject and no students. In the lattice concept, C_1 is the infimum (also called *meet*) of the concepts C_2 and C_3 , while C_4 is their supremum (also called *join*).

The lattice with the succinct labelling can be used to answer some questions efficiently.

An object has a property if the object label is not higher than the property label. To find all objects that have given property, one needs to start at the node labelled by the property and traverse the lattice below the property label. For example, English is liked by Arnav and Aarti. Likewise, to find all properties of a given object, one needs to traverse the lattice above the node labelled with the object.

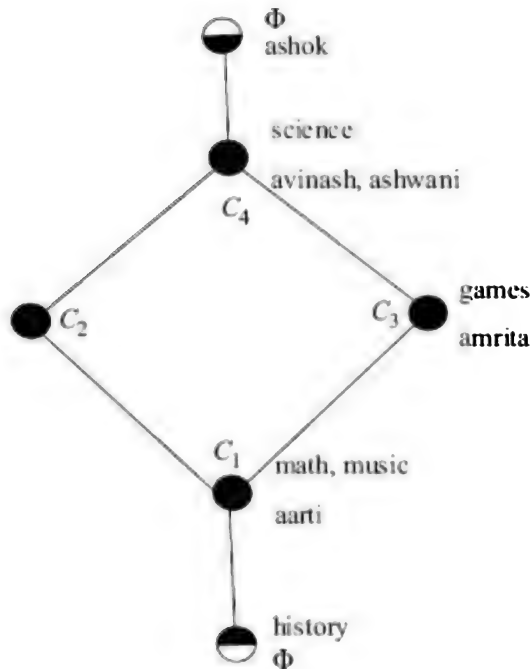


FIGURE 14.35 The concept lattice for the context given in Table 14.2. Observe that there is a property “history” that does not hold for any object.

To find the common properties shared by two objects, one has to find the *join* or the lowest common subsumer. In our example, to find what properties (subjects) are shared (liked) by Arnav and Amrita, we traverse upwards from their respective nodes C_2 and C_3 , to concept C_4 which has the intension {science}.

Likewise, to find objects that share two properties, one has to traverse downward from the nodes labelled by the two properties to the highest common subsumee (or *meet*) of the two nodes. In Figure 14.35, to find the students who like both English and Games, we inspect their meet, the node C_1 which is the extension {aarti}.

Observe that,

$$\begin{aligned} \text{int}(C_2) &= \{\text{english, science}\} \\ \text{int}(C_3) &= \{\text{science, games}\} \end{aligned}$$

and their union is {english, science, games} which is smaller than the intension of their infimum C_1 {math, english, science, games, music}. This means that the union of the intensions of two concepts may be smaller than the intension of their infimum. Likewise,

$$\text{ext}(C_2) \cup \text{ext}(C_3) \subseteq \text{ext}(C_4)$$

It is also the case that the union of the extensions of two concepts may be smaller than the extension of their supremum. However, the intersection of intensions is equal to the intension of the supremum, and the intersection of extensions is equal to the extension of the infimum.

14.8.1 From Context to Concept Lattice

Given a set of objects $O \subseteq G$, a concept can be formed by (possibly) extending the set to include other objects that share the corresponding properties. That is, the concept is computed as $(\text{obj}(\text{att}(O)), \text{att}(O))$. But to find all concepts, one would have to wade through all the subsets of G , which are exponentially many.

We sketch an algorithm to find the concepts and construct the concept lattice at the same time. The input to the procedure is the context K . The task is both to discover the formal concepts and structure them into a lattice. We have mentioned that concepts can be found by inspecting all subsets of G or M , but that would be a computationally expensive brute approach. Many approaches to building concept lattices have been developed (Waltchev and Missouri, 2001) (Baixeries et al., 2009). One approach starts with a single object from the context and incrementally adds more objects one by one and carries out structural updates to the lattice (Godin et al., 1995). We look at another approach in which the algorithm begins with the top level concepts and recursively finds its *successors* by adding a property to the intension of the current concept (Choi, 2006).

Let $\text{nbr}(m)$ be a function that returns the objects that have the property m . This can be done by scanning the column in K labelled m .

Let C be a concept in $\mathcal{B}(G, M, R)$. Then, $\text{int}(C)$ is the set of properties in C . Let m be a property in M that does not belong to $\text{int}(C)$. Let E_m be

the set of objects defined as,

$$E_m = \text{ext}(C) \cap \text{nbr}(m)$$

If this is not empty, it means that there is at least one object that is part of the concept C and has property m . Furthermore, it is the case that E_m is closed. That is,

$$\text{obj}(\text{att}(E_m)) = E_m$$

The proof is as follows. By definition, $E_m \subseteq \text{obj}(\text{att}(E_m))$. To show the reverse $\text{obj}(\text{att}(E_m)) \subseteq E_m$, observe that,

$$\begin{aligned} \text{obj}(\text{int}(C) \cup \{m\}) &= (\bigcap_{j \in \text{int}(C)} \text{nbr}(j)) \cap \text{nbr}(m) = \text{ext}(C) \cap \text{nbr}(m) \\ &= E_m \end{aligned}$$

Since E_m is closed, it follows that $(E_m, \text{att}(E_m))$ is a formal concept.

Consider in our example the top level concept: $\top = (\text{avinash}, \text{arnav}, \text{aarti}, \text{amrita}, \text{ashwani}, \text{ashok}, \{\})$ for the context in Table 14.2. Extending with each of the properties we have $E_{\text{math}} = \{\text{aarti}\}$, $E_{\text{english}} = \{\text{arnav}, \text{aarti}\}$, $E_{\text{science}} = \{\text{avinash}, \text{arnav}, \text{aarti}, \text{amrita}, \text{ashwani}\}$, $E_{\text{games}} = \{\text{aarti}, \text{amrita}\}$, and $E_{\text{music}} = \{\text{aarti}\}$.

The corresponding (closed) property sets are,

$$\begin{aligned} \text{att}(E_{\text{math}}) &= \{\text{math}, \text{english}, \text{science}, \text{games}, \text{music}\} \\ \text{att}(E_{\text{english}}) &= \{\text{english}, \text{science}\} \\ \text{att}(E_{\text{science}}) &= \{\text{science}\} \\ \text{att}(E_{\text{games}}) &= \{\text{science}, \text{games}\} \\ \text{att}(E_{\text{music}}) &= \{\text{math}, \text{english}, \text{science}, \text{games}, \text{music}\} \end{aligned}$$

For a closed set of properties $(X = \text{att}(C)) \subset M$, the set of remaining attributes $\text{res}(X)$ that can be added to create a new concept is defined as,

$$\text{res}(X) = \{m \in M \setminus X \mid (\text{obj}(X) \cap \text{nbr}(m)) \neq \{\}\}$$

It is possible that for two distinct properties $n, m \in \text{res}(X)$ the corresponding objects set is identical. Having identical object sets induces an equivalence relation that partitions $\text{res}(X)$.

$$\text{res}(X) = S_1 \cup S_2 \cup \dots \cup S_t$$

Each such partition will correspond to a formal concept and a *possible* child of the concept C . Let the set of equivalence classes be called the $\text{AttrChild}(X) = \{S_1, S_2, \dots, S_t\}$. We assume that the subscript of S is mapped to the attributes that are in that partition. For the above example $X = \{\}$,

$$\begin{aligned}\text{AttrChild}(\{\}) &= \{ S_{\text{math,music}} = \{\text{math, english, science, games, history}\}, \\ &\quad S_{\text{english}} = \{\text{english, science}\}, \\ &\quad S_{\text{science}} = \{\text{science}\}, \\ &\quad S_{\text{games}} = \{\text{science, games}\} \end{aligned}$$

For each $S_i \in \text{AttrChild}(X)$, the pair $(\text{obj}(S_i \cup X), (S_i \cup X))$ is a formal concept. In the above example, $X=\{\}$ and the corresponding child concepts of \top are,

$$\begin{aligned}C_{\text{math,music}} &= (\{\text{aarti}\}, \{\text{math, english, science, games, history}\}) \\ C_{\text{english}} &= (\{\text{arnav, aarti}\}, \{\text{english, science}\}) \\ C_{\text{games}} &= (\{\text{aarti, amrita}\}, \{\text{science, games}\}) \\ C_{\text{science}} &= (\{\text{avinash, arnav, aarti, amrita, ashwani}\}, \{\text{science}\}) \end{aligned}$$

Let $\text{Succ}(X)$ be a set of attribute sets that correspond to the set of successors of concept C in the concept lattice. Given that by definition there cannot be two successors C_k and C_j such that $C_k \prec C_j$, some of the elements in $\text{AttrChild}(X)$ may not correspond to the successors of C . They will move further down in the lattice. As defined above, $C_k \prec C_j \equiv S_j \subset S_k$. Since in the above example,

$$S_{\text{science}} \subset S_{\text{math,music}}$$

and

$$S_{\text{science}} \subset S_{\text{english}} \subset S_{\text{math,music}}$$

and

$$S_{\text{science}} \subset S_{\text{english}} \subset S_{\text{math,music}}$$

only C_{science} qualifies to be a successor of the top element $\top = (\{\text{avinash, arnav, amrita, amnta, ashwaru}\}, \{\})$.

A simpler method of deciding whether an equivalence class in $\text{AttrChild}(X)$ corresponds to a successor of X is as follows. Let E be the set of attributes associated with the equivalence class S_E . E is the set of attributes that extend $\text{obj}(X)$ by the same set of new objects. If $|\text{obj}(S_E \cup X)| > |E|$ then S_E is *not* a successor of X . In our example above, the set $\{\text{english, science}\}$ is larger than $\{\text{english}\}$, and therefore cannot be the intension of a successor of \top . As one can see, in the example S_{science} is the only equivalence class for which the set of attributes $\{\text{science}\}$ is not larger than the defining set of attributes $\{\text{science}\}$.

At this stage, the lattice contains two elements, \top and its only successor C_{science} . Next, we recursively add each of the other four attributes—*math*, *music*, *english* and *games*—one by one to the attributes of C_{science} , which in this case is the single attribute *science*. Then we construct its children (which are the same as the ones discarded above), and select the next two successors C_{games} and C_{english} . Finally,

$C_{math,music}$ is added as a successor twice for each of C_{games} and $C_{english}$. The reader is encouraged to apply the set size test described above for these equivalence classes.

The high level algorithm adapted from (Choi, 2006) is given in Figure 14.36 below.

```

GaloisLatticeConstruction(g,m,r)
1  Compute top concept c = (g, att(g))
2  Compute Child(c)
3  Succ(c) ← ( )
4  enqueue(c, q)
5  while not empty(q)
6    do c ← dequeue(q)
7    x ← int(c)
8    Compute AttrChild(x) = (s1, s2, ..., sK)
9    for i = 1 to K
10     do Let di be the set of defining attributes for si
11     if |si ∪ x| = |di|
12     then Let k = (obj(si ∪ x), (si ∪ x))
13     if k ∉ q
14     then Compute Child(k)
15     enqueue(k, q)
16     Succ(k) ← ( )
17     Succ(c) ← cons(k, Succ(c))
18  return top=(g, att(g))

```

Fig 14.36 The procedure to find the concepts for the given context (g,m,r) and construct the lattice. With every concept c , there is a list of child concepts in $Child(c)$ and a list $AttrChild(x)$ that contains the equivalence classes of the attributes of the children. The algorithm returns a pointer to the top concept.

14.9 Conceptual Graphs

Perhaps the most general form of representing logical sentences in the form of a semantic network is the formalism of Conceptual Graphs introduced by John F. Sowa (1984; 2000; 2009).

Sowa traces the origin of graphical notation of concepts to Charles Sanders Peirce who was “*searching for a graphic notation, similar to the notations used in organic chemistry, that would more clearly show “the atoms and molecules of logic.”*” (Sowa, 2006). Around the same time, Gottfried Frege had also developed a treelike notation (Frege, 1879). The earliest graphs developed by Peirce were relational graphs, capable of representing only conjuncts of existentially qualified variables. In 1897, however, he invented a remarkable technique that made it possible to represent all FOL formulas graphically (Peirce, 1909). This technique involved the drawing of an oval to represent negation of whatever was inside the oval. Consider the example (from (Sowa, 2006; 2009))—“*If a farmer owns a donkey then he beats it*”. In FOL, we would write the sentence as,

$$\forall x \forall y ((\text{Farmer}(x) \wedge \text{Donkey}(y) \wedge \text{Owns}(x, y)) \supset \text{Beats}(x, y))$$

This requires the representation of the universal quantifier and the implication operator. However, with the introduction of negation, the sentence can equivalently be written as,

$$\neg \exists x \exists y (Farmer(x) \wedge Donkey(y) \wedge Owns(x, y) \wedge \neg Beats(x, y))$$

which Peirce could draw with the help of two ovals as shown in Figure 14.37.

The existential graphs of Peirce are the starting point for the conceptual graphs of Sowa. The idea of constructing networks of nodes to capture relationships in a systematic manner has been explored by many researchers. The CD structures of Schank are also networks built on a well defined vocabulary of CD actions and states (see Chapter 13). Most other efforts use the words from natural languages to build their networks. One notable example is the network representations used in the *Semantic Network Processing System* (SNePS) (Maida and Shapiro, 1982), (Shapiro and Rapaport, 1986), (Shapiro, 2000). Figure 14.38 shows the SNePS representation for “*Sue thinks that Bob believes that a dog is eating a bone.*”

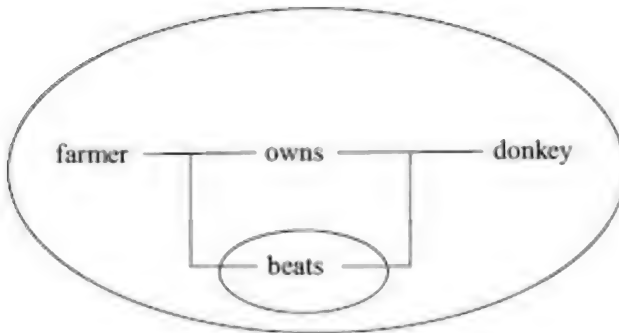


FIGURE 14.37 Peirce’s existential graph for the sentence “*If a farmer owns a donkey, then he beats it.*” The oval is a negation of all that it encloses.

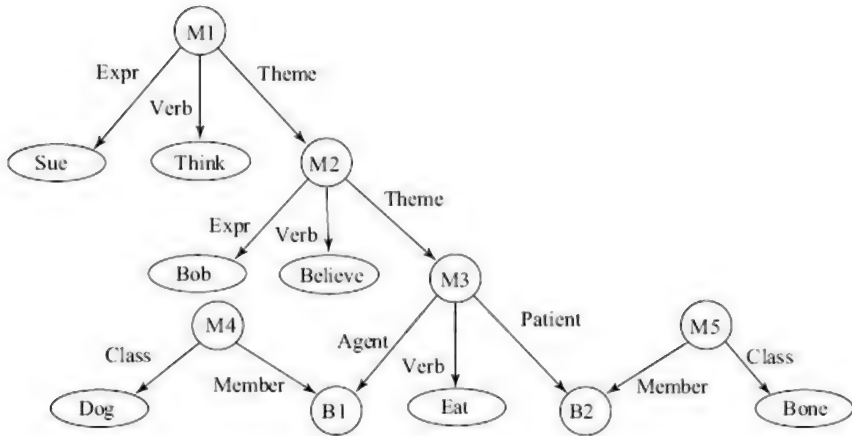


FIGURE 14.38 The sentence “Sue thinks that Bob believes that a dog is eating a bone” as represented in SNePS (figure adapted from (Sowa, 2006)).

The nodes labelled M represent propositions. Observe that a proposition can be the subject matter of another proposition; something that we have seen needs a process of reification in FOL. Proposition M_1 asserts that Sue *thinks* that M_2 is true. M_2 says that Bob *believes* that M_3 is true³². M_3 says that B_1 ate B_2 , M_4 says that B_1 is a dog, and M_5 says that B_2 is a bone. The specific relation between two nodes is specified by labels on the edges that are not shown here.

Sowa solved the problem of having to label edges by introducing them as *relation* nodes. The same sentence represented in a conceptual graph is depicted in Figure 14.39.

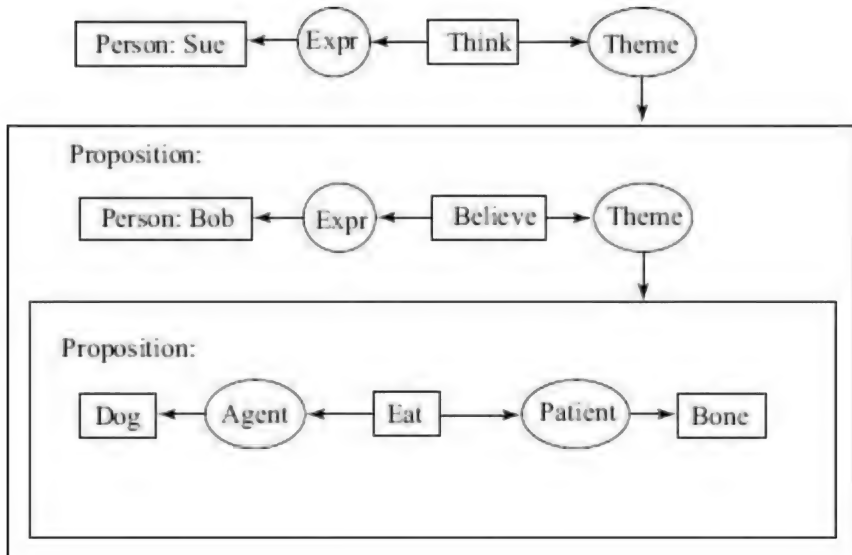


FIGURE 14.39 The sentence “Sue thinks that Bob believes that a dog is eating a bone” as represented in a conceptual graph (figure adapted from (Sowa, 2006)).

The relations are the same. Sue is the *experiencer* of think, and the *theme* of her thinking is the proposition in the box. Conceptual Graphs (CGs) are semantic networks that have a logical and linguistic basis, and which can handle quantified variables along with constants. CGs are bipartite graphs with two kind of nodes with directed arcs across them.

On one side are nodes that stand for elements of a domain, along with their class labels. These are drawn as boxes. The elements may be named, unnamed, more than one, or all the elements of a class. The class labels or concepts are organized into an abstraction hierarchy, represented as a lattice. It might be worth noting that the elements of the domain include not just words corresponding to noun phrases, but also verbs and adjectives. With reference to the Conceptual Dependency theory (see Chapter 13), the domain contains PPs, ACTs, PAs and AAs.

On the other side of the bipartite graph are nodes representing relations between elements of the domain. These nodes are drawn as ovals. These relations are themselves ordered into an abstraction hierarchy. While the concepts can stand by themselves, the relation nodes need the concept nodes that they relate to exist in a knowledge base. The edges that connect relations to concepts are said to *belong* to the relation node and *attached* to the concept nodes.

The knowledge base is a set of CGs describing situations. The simplest facts can be concepts that assert the existence of elements of a class. Others can describe, like conceptualizations in the CD theory, relations between things, attributes and events. The CGs have a convenient symbolic notation that could be used in computer programs. We represent the concept nodes in square brackets, for example (Person: Sue) or [Dog]. The relation nodes are enclosed in round brackets, as in (Agent) or (Theme).

In general, a concept (node) is made up two entities, a *type* which is a label that specifies the class of the concept, and a *referent* that denotes an individual of that class. For example, in the concept [Person: Sue] the type is “Person” and the referent is “Sue”. The special case when the referent is omitted is a short form for a concept with a referent “ \exists ” that is like an existential quantifier. Thus [Dog] is a short form for [Dog: \exists] and read “there exists a dog”. One can also say [Dog: 3 Snoopy] to assert “there exists a dog named Snoopy”. If one wanted to say for example that *all* dogs are intelligent then one would have used the concept [Dog: \forall]. Other forms of referents are as follows.

- [Dog: #] is read as “the dog”
- [Person: #she] is a pronoun as in “she is eating a bun”. Other pronouns are “#it”, “#you”, etc.
- [Student: #3] One can also use the “#” with an index, for example “the 3rd student”.
- [Student: {arnav,aarti,amrita}] has as referents a set of named entities of a particular class.

- [Student: {*}] refers to a set of entities whose elements are not named.
- [Student: {*}@4] refers to a set of 4 unnamed students.
- [Parent: @2] refers to the 2 parents that a person may have.
- [Boy Abheek *x] and [Boy ?x] are a pair of concepts that may occur in different graphs in a context but that refer to the same entity. The marker "x" establishes the connection.

Concepts can stand by themselves, but relations need the concepts that the relations relate. The simplest such graph embodies a dyadic relation, depicted by the *star graph* shown below,

[Concept1] → (relation) → [Concept2]

The following two graphs are identical and read "a cat is on a mat",

[Cat] → (On) → [Mat]

or

[Mat] ← (On) ← [Cat]

The following would be the title of the well-known children's book by Dr. Seuss.

[Cat: #] → (In) → [Hat: #]

A triadic relation may be represented as follows,

[Book: Asterix & Obelix] ← (Authors-2) –
 ←1– [Person: Goscinny]
 ←2– [Person: Uderzo]

which in the graphical is shown in Figure 14.40.

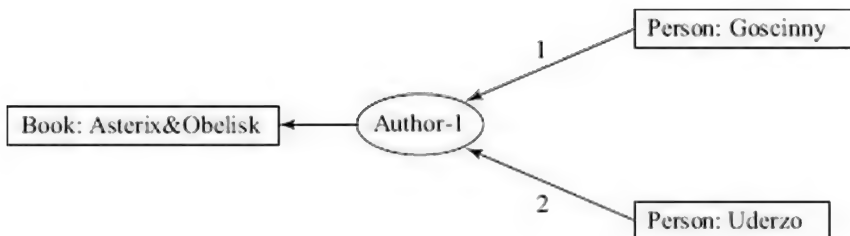


FIGURE 14.40 A triadic relation in a conceptual graph.

Every n -ary relation has a *signature* that is made of the types of the concepts it relates. The signatures of the three graphs described above are <Cat, Mat>, <Cat, Hat>, and <Person, Person, Book>. It is a convention that the arrows are directed away from all but the last concept in the signature. If these are more than one then they are numbered

increasingly from left to right. The last concept in the signature has an arrow directed towards it from the relation.

Figure 14.41 shows a larger example of a CG.

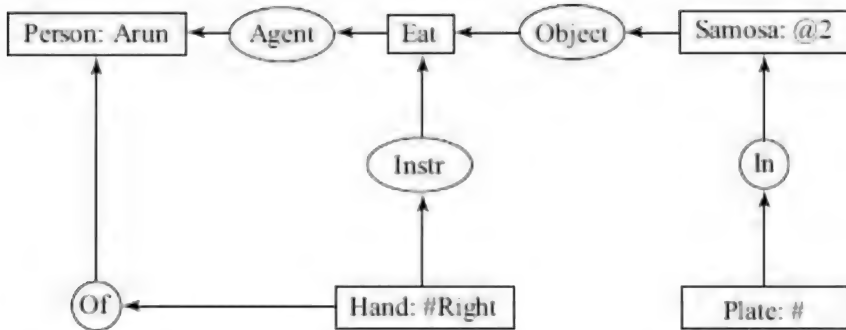


FIGURE 14.41 The conceptual graph for “Arun ate two samosas that were in the plate with his right hand.”

Observe that all relations in the above CG are dyadic, and therefore no numbering is necessary. This could be represented as,

```
[Eat] -
  (Agent) → [Person: Arun *x]
  (Instr) ← [Hand: #Right] → (Of) → [Person: ?x]
  (Object) ← [Samosa @2] ← (In) ← [Plate: #].
```

We could have also written this as conjunction written as,

```
[Proposition:
  [Eat *y] → (Agent) → [Person: Arun *x]
  [Eat ?y] ← (Instr) ← [Hand: #Right] → (Of) → [Person: ?x]
  [Eat ?y] ← (Object) ← [Samosa @2] ← (In) ← [Plate: #]
]
```

Conceptual graphs allow conceptual graphs to be reified and hence are more powerful than first-order logic. This was illustrated in the example about Sue, Bob and the dog above. One can reify a CG by using a monadic relation to create a new concept. For example we can say,

```
[Situation: [Eat] -
  (Agent) → [Person: Arun *x]
  (Instr) ← [Hand: #Right] → (Of) → [Person: ?x]
  (Object) ← [Samosa @2] ← (In) ← [Plate: #]]
```

and one could use this elsewhere, for example in “the situation is that Arun’s mother saw Arun eating two samosas on a plate with his right hand”

```
[Situation:
  [Mother *z] ← (Of) ← [Person: Arun *x]
  [Mother ?z] → (See) → [Situation: [Eat] -
    (Agent) → [Person: Arun ?x]
    (Instr) ← [Hand: #Right] → (Of) → [Person: ?x]
    (Object) ← [Samosa @2] ← (In) ← [Plate: #]]]
```

Here is another example, “The teacher thinks that all students are honest”,

```
[Teacher: #] ← (Expr) ← [Belief] → (Theme) →
  [Proposition: [Student:∀] → (Is) → [Honest]]
```

Once a proposition has been created, then one can also create its negation. The following CG states that “The teacher thinks that it is not the case that all students are honest”

```
[Teacher: #] ← (Expr) ← [Believe] → (Theme) →
  ¬[Proposition: [Student:∀] → (Is) → [Honest]]
```

Both Sowa and, before him, Peirce were motivated by having a graphical mechanism for reasoning. There have been several attempts to devise graph manipulation operators (see for example (Peirce, 1909)). It is not entirely clear whether a complete deduction system has emerged, but we refer the reader to (Sowa, 2006; 2009) for more details. Meanwhile, conceptual graph are also semantic networks and we look at an approach for semantic retrieval using CGs below.

14.9.1 Efficient Matching with CGs

One of the uses that Conceptual Graphs can be put is for semantic retrieval, in which the query is a CG that needs to be matched to the appropriate subgraph in the knowledge base in the form of a CG. We look at an approach by Galia Angelova (2009) that converts *simple conceptual graphs* into a minimal finite state automaton. A simple conceptual graph is a subset of FOL that employs only the existential quantifier and the conjunction operator without negation.

A Simple Conceptual Graph may be described using the *vocabulary* or *support* as follows.

Definition

A support is a 4-tuple $S = (T_C, T_R, I, \tau)$ where,

- T_C is a finite, partially ordered set of distinct concept types. The partial order defines a taxonomy with \top as the *universal* type that is the supremum of the concept lattice. For the sake of completeness, we also define the *absurd* type \perp that forms the infimum of the concept lattice.

- T_R is a finite, partially ordered set of distinct relation types. The two sets T_C and T_R are disjoint. Each relation $R \in T_R$ has arity 2 (in the case of SCGs), and holds between two elements of the concept type. For each R , a pair of concepts $(C_{1R}, C_{2R}) \in T_C \times T_C$ defines the highest (most general) type of concepts that may be related by R . All pairs (C_{1R}, C_{2R}) are called *star graphs*. If $R_1 \leq R_2$ then $C_{1R_1} \leq C_{1R_2}$ and $C_{2R_1} \leq C_{2R_2}$. Like in the case of the concept hierarchy, we also define the supremum and infimum for the relation types.
- I is a set of distinct individual markers that refer to a specified individual. The generic marker $*$ $\in I$ refers to an unspecified individual.
- τ is a mapping from I to T_C specifying instances of concepts.

The support or vocabulary encompasses all concepts, individuals and relations that are used to build the simple, conceptual graphs. Figure 14.42 illustrates the concept and relation lattices that form the support for an example, conceptual graph domain.

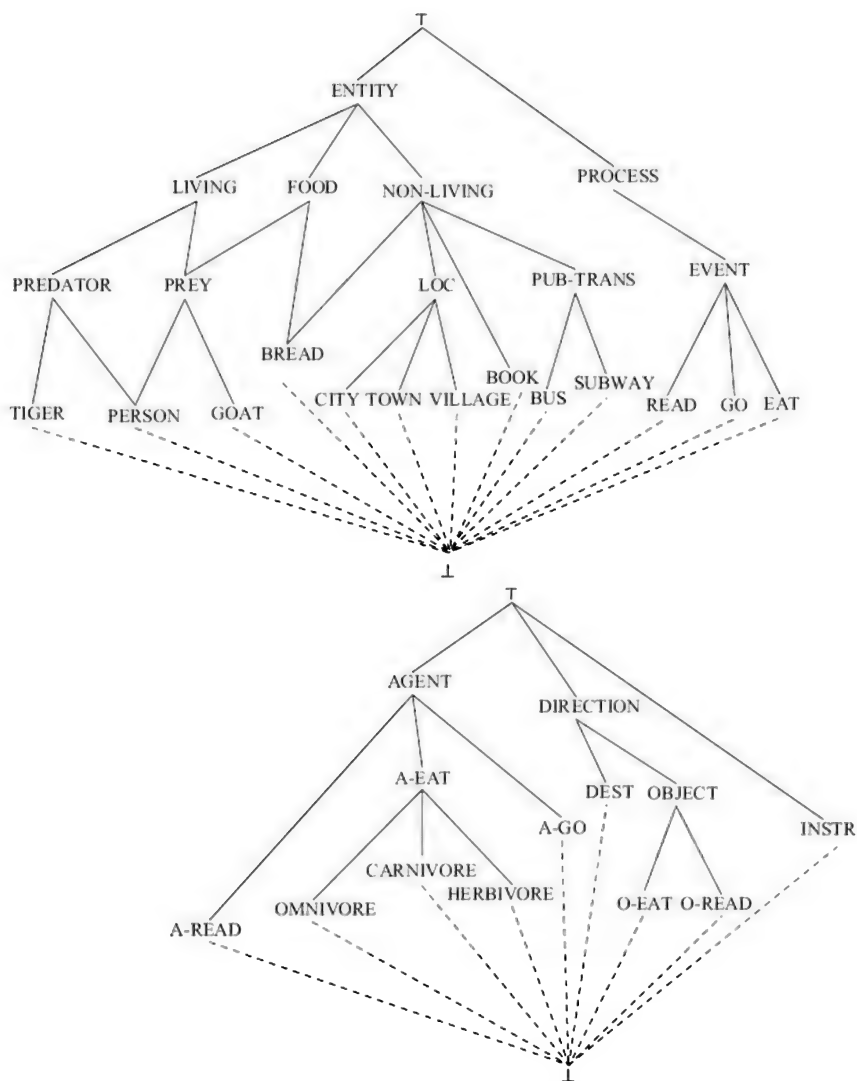


FIGURE 14.42 An example concept lattice (top) and a relation lattice (below) that forms the support for CGS.

Some of the star graphs for the support of Figure 14.42 are given below.


```

[Event] → (Agent) → [Living]
[Eat] → (A-Eat) → [Living]
[Go] → (A-Go) → [Living]
[Go] → (Dest) → [Loc]
[Eat] → (Carnivore) → [Predator]
[Read] → (A-Read) → [Human]
[Go] → (Dest) → [Loc]
[Read] → (O-Read) → [Book]
[Eat] → (O-Eat) → [Food]
[Go] → (Instr) → [Pub-Trans]

```

Given two SCGs, G and H defined on the same support, we can define an injective projection as follows (Mugnier, 1995). A mapping $\pi: G \rightarrow H$ yields a graph $\pi G \subseteq H$, such that πG is isomorphic to G , and for each concept c in G , πc is a concept in πG (and H) where $\text{type}(\pi c) \leq \text{type}(c)$. The SCG G is called the *injective generalization* of πG . If G is a query subgraph then the projection extracts all subgraphs in the knowledge base that are specializations of G . In general, the graph isomorphism problem is NP-complete. However, for simple conceptual graphs, one can devise polynomial time algorithms.

The algorithm proposed in Angelova (2009) preprocesses the knowledge base to construct a finite state automaton that captures all possible queries that have answers in the knowledge base. The finite state automaton has a number of acceptance states with markers that represent the answer to the query. The words of the language accepted by the FSA constitute of (Angelova and Mihov, 2008),

1. all SCGs in the knowledge base,
2. all their conceptual subgraphs³³, and
3. all the injective generalizations of the above two.

Consider the following CG for the sentence "John took the subway to Coney island."

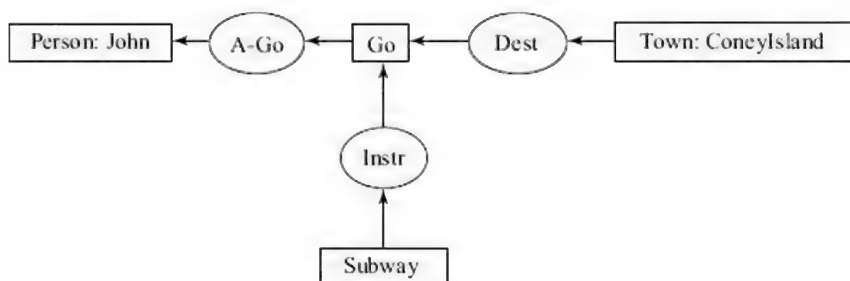


FIGURE 14.43 The conceptual graph for "John took a subway to Coney Island."

The following are the subgraphs and their injective generalizations, which represent possible queries.

```

1. [Go] - /* the input */
   (A-Go) → [Person: John]
   (Instr) ← [Subway]
   (Dest) ← [Town: ConeyIsland]

2. [Person: John] ← (A-Go) ← [Go] /* subgraphs */
3. [Go] → (Instr) → [Subway]
4. [Go] → (Dest) → [Town: ConeyIsland]
5. [Person: John] ← (A-Go) ← [Go] → (Instr) → [Subway]
6. [Person: John] ← (A-Go) ← [Go] → (Dest) → [Town: ConeyIsland]
7. [Subway] ← (Instr) ← [Go] → (Dest) → [Town: ConeyIsland]

8. [Person] ← (A-Go) ← [Go] /* generalizations of star graphs */
9. [Living] ← (A-Go) ← [Go]
10. [Go] → (Instr) → [Pub-Trans]
11. [Go] → (Dest) → [Town]
12. [Go] → (Dest) → [Loc] /* and their combinations into larger
                               graphs... */

```

Observe that the process of generalization stops with the signatures of the relations.

```

[Go] → (A-Go) → [Living]
[Go] → (Dest) → [Loc]
[Go] → (Instr) → [Pub-Trans]

```

Each star graph and its generalization can be expressed as a triple. Some of the triples above are (Go, A-Go, Living), (Go, A-Go, Person), (Go, Dest, Town), and so on. These triples are sorted alphabetically and form three consecutive edges of the FSA, which will be traversed with the tree inputs. Given that every acyclic automaton can be described by the finite list of words belonging to that language, once we have a collection of such triples and their allowed concatenations, we have essentially defined the FSA. When we have two triples, for example, (Go, A-Go, Person) and (Go, Dest, Town) we have different possible interpretations, one in which Go refers to the same action, and one in which it refers to two different ones. These interpretations should be marked differently, and Angelova proposes the construction of equivalent classes in which such identities are distinguished. Not all partitions induced are relevant though, and the relevant ones are marked. The corresponding subgraphs become the objects for retrieval, and the markers are attached to the corresponding final states in the FSA. The same marker may appear as a label for a different final state in the FSA, since the corresponding CG may satisfy more than one query. What remains is to construct a minimal FSA with the set of triples. We will not go into the details here. The interested reader is referred to (Angelova and Mihov, 2008) and (Angelova, 2009).

14.10 Discussion

In this chapter, we have looked at various approaches designed to speed up the accessing of related information. In principle, one could have used first order logic with reification to represent most things that are of interest. However, it is when we need to find connections between different facts that the prospect of structured representations comes to the fore. The notion of the schema has been investigated by many people including psychologists, philosophers and computer scientists. The notion of pulling together things into chunks and representing the reified composite elements offers a way of cutting down upon the sheer number of facts that we often want to abstract away from. The concept of *Gestalt* refers to the *form forming ability* of our senses,³⁴ of being able to focus on the whole rather than on the sum of the parts, and is a key to understanding the schema. The other feature needed for rapid access to related concepts is via networking of concepts into a semantic network. As a consequence, representation of large amounts of knowledge involves the construction of inheritance and abstraction hierarchies. A host of efforts have been made to achieving these ends, some in which the user is required to specify the agglomerations and the connections, and others in which the user just specifies a description the structure follows. It can safely be said that the construction of knowledge structures and reasoning with them is going to occupy AI researchers for a while.



Exercises

1. Of the different meanings of the sentence "Time flies like an arrow", which one occurs to you? Show the nodes that will be activated in the network in Figure 14.10. Add new nodes and edges where required.
2. "A tomato is a vegetable. A tomato is a fruit. A tomato can be fried. A vegetable is a something that can be fried. Fruits cannot be fried. Something that can be fried is a kind of something that can be cooked. A vegetable is something that can be cooked."

Remove the redundant statements from the above set. What are the *credulous extensions* of the resulting set of statements?

3. Write an algorithm to determine whether an edge in an inheritance graph is admissible or not with respect to an input node A.
4. Given an inheritance graph Γ , an input node A, and a labelling of each edge saying whether it is admissible or not with respect to A, write an algorithm to find all extensions of the graph.
5. Allow the user to create an inheritance hierarchy graphically (or read from a file). Given a taxonomy, answer the question $a \rightarrow p$? Create extensions, identify the preferred extensions.
6. What are the credulous extensions with respect to a' of the inheritance graph in Figure 14.22. Which of these are preferred?

7. Given the two concepts,

$$d \doteq [\text{AND } [\text{FILLS :FanOf beckenbauer}] [\text{FILLS :FanOf theKaiser}]]$$

and

$$e = [\text{EXISTS 2 :FanOf}]$$

can we conclude that $d \sqsubseteq e$? What does the structure mapping algorithm say? Explain your answer. [Hint: What do tokens map to?]

8. Express the following sentences in FOL as well as in Description Logic.
- A RichFooTween is a ten-year old person who has at least two friends and who is a football fan and all whose aunts are rich.
 - A JailHoParty is political party whose members are all unemployed with all siblings politicians, and which has at least two PM candidates and all whose ministers have at least two bodyguards, and two court cases.
9. Given the following two statements (Brachman and Levesque, 2004),

```

marianne → [AND Person Female]
ellen → [AND [EXISTS 1 :Child]
          [ALL :Child [AND [FILLS :Pediatrician marriane]
                           [ALL :Pediatrician Scandainavian]]]]

```

Show how the information about Marianne is augmented.

10. Express the following sentences in Description Logic.
- Sachin plays for Mumbai Indians and has more than one million fans.
 - Sucheta is a faculty member.
 - The ABC University employs Sucheta.
 - Sucheta has at least two cousins, and all her cousins are fans only of cricketers and are all fans of Sachin.
 - ABC University is a university that employs only Indian citizens who have a PhD degree. What are the most specific concepts that the individuals Sachin and Sucheta belong to? Explain the procedure used to arrive at the answers.
11. Given the following information about the given set of individuals,

```

baba → Politician
cP → [AND Party [ALL :Leaders Rich] [FILLS :Leader baba]]
fFam → [AND Family [FILLS :Member baba]
        [ALL :Member Famous] [ALL :Member Dynasty]]
backV → [AND [FILLS :Voted cP] [ALL :Voted Corrupt]]
sachin → [AND [FILLS :Team bombay] Famous [EXISTS 100000 :Fan]]
sucheta → [AND [EXISTS 2 :Cousin]
           [ALL :Cousin [AND [FILLS :FanOf sachin] [ALL :FanOf Cricketer]]]]

```

Collect all information about each individual in one place, using the

propagation algorithm. Show each step from the algorithm clearly.

12. Given two descriptions $d_{sub} = [D-EXISTS\ n\ r\ d]$ and $d_{super} = [D-EXISTS\ m\ r\ e]$, write a structure matching rule that can be used to answer the question whether ($d_{sub} \sqsubseteq d_{super}$).
13. One can extend the description logic from Section 14.7 (with the ALL, EXISTS, FILLS and AND operators) to include an operator $[D-EXISTS\ n\ r\ d]$ which can be used to say that there are n role fillers from class d . For example, to describe the set of all people with at least two children who are girls one would say $[D-EXISTS\ 2\ :Child\ Girl]$. Give the formal semantics of this new operator.

An alternative is to use the existing operators and introduce a new role named *GirlChild*. Discuss the pros and cons of the two approaches.

14. The structure matching rule looks for corresponding components in d_{sub} and d_{super} . We can extend the set of structure matching rules to take more than one component as argument. Consider the task of handling $[D-EXISTS\ 1\ :Sibling\ Parent]$ and $[AND\ [EXISTS\ 2\ :Sibling]\ [ALL\ :Sibling\ Parent]]$ which can be rewritten as $[AND\ [D-EXISTS\ 2\ :Sibling\ Thing]\ [ALL\ :Sibling\ Parent]]$. Write a rule that will take $d_{super} = [D-EXISTS\ n\ r\ d]$ and $d_{sub} = [AND\ [D-EXISTS\ m\ r\ e]\ [ALL\ r\ f]]$, write a rule to answer the question whether ($d_{sub} \sqsubseteq d_{super}$). Discuss the pros and cons of using such a rule.
15. Is any of the two classes—*realMusicLover* and *beenutShroeder*—defined in the chapter, subsumed by the other?
16. Description Logic 1. Write a program to create a knowledge base of concepts. Allow the user to create new concepts from old by “extending” them with more constraints, and using AND to combine concepts. The descriptions may have to be converted into normal form. Implement the Structure Matching algorithm.
17. Description Logic 2. Given a set of concepts read from a file, construct a Taxonomy. Display the graph on a screen and allow users to read concept definitions attached to nodes on demand. Accept a new concept and classify it into the taxonomy.
18. Description Logic 3. A-Box reasoning. Given a database of facts, extract concepts that individuals belong to, including the propagation to ferret out information. Attach them to the most specific concept in the Taxonomy. Accept new information of the kind ($a \rightarrow d$) and propagate the information. The system should be able to answer queries like (1) $c \rightarrow d$, (2) find all classes that an individual c belongs to, and (3) find all individuals that belong to a class
19. Write an OWL statement to assert that the classes *9symphonies* and *symphoniesOfBeethoven* defined in the chapter represent the same sets of elements.
20. Given a set of objects O , show that (O'', O') is a formal concept where $O' = att(O)$ and $O'' = obj(att(O))$.

21. If the properties in Table 14.1 have the following correspondence —1:preying, 2:flying, 3:bird, and 4:mammal, label the names of objects by corresponding animals. (See also [Wolff, 1993]).
22. In the context shown in Table 14.1 renamed by the previous exercise, insert the animal “bat” with the properties “mammal” and “preying”. Extend the concept lattice of Figure 14.34 to reflect the new context.
23. Given a set of natural numbers 1–12 and the properties $P = \{\text{composite, even, odd, prime, square}\}$, construct a formal context and the corresponding concept lattice³⁵.
24. Given a context in the form of a table, implement the FCA algorithm to build the concept lattice. Display the lattice graphically. Given a concept C , one should be able to view the extent and the intent. *Extension*: Given a numeric attribute, a user should be able to select ranges with memberships to these ranges as attributes.
25. Use the negation and the conjunction operator to express disjunctions in conceptual graphs. Show how the sentence “The cat is either on the mat or in the kitchen.”
26. Show how conceptual graphs can be used to express IF-THEN kind of assertions.
27. The $O\text{-}Eat$ relation says that for the event (act), Eat the object is $Food$. Add specializations of $O\text{-}Eat$ to cater to the *type* of food herbivores and carnivores eat.

¹ <http://en.wikipedia.org/wiki/Phenomenalism>

² Chunking—put together parts and give it a name.

³ The term *atom* is derived from the Greek term *átomos* meaning, uncuttable or indivisible, coined by Democritus around 450 bc.

⁴ Or sometimes, like Sherlock Holmes, consciously avoid knowledge that is not relevant to one’s goals. In *A Study in Scarlet* (by Sir Arthur Conan Doyle) Holmes says that he would now do his best to forget this fact (that the Earth goes round the Sun) as “*it would not make a pennyworth of difference to me or my work.*” ... “*I consider that a man’s brain originally is like a little empty attic, and you have to stock it with such furniture as you choose.... A fool takes in all the lumber of every sort that he comes across, so that the knowledge which might be useful to him gets crowded out, or at best is jumbled up with a lot of other things so that he has a difficulty in laying his hands upon it. Now the skilful workman is very careful indeed as to what he takes into his brain-attic.*”

⁵ http://en.wikipedia.org/wiki/Critique_of_Pure_Reason

⁶ See <http://penta.ufrgs.br/edu/telelab/2/war-of-t.htm>

⁷ By concrete we do not mean physical, just something that is an element of the domain of discourse.

- ⁸ See <http://www.tolweb.org/tree/>
- ⁹ One assumes that fillers of slots are pointers to the constituent frames. This makes each frame a compact structure. If one were to use the frames themselves as fillers, the structures would become quite unwieldy, considering that the aggregation hierarchy could be quite deep.
- ¹⁰ In our times, an SMS or an email would be the medium.
- ¹¹ On this network, the answer would be “Aeden is the cousin of Arushi’s mother”.
- ¹² In his 1977 paper, Charniak said in the concluding section “*I would estimate that the version which will handle all of the examples herein is six months off, but previous experience tells me that such estimates are likely to be too ambitious by a factor of two or three.*” In fact, the AI community is still on the job!
- ¹³ Teenagers can respond to a bewildering variety of questions with the answer “Simply” .
- ¹⁴ A jar of pickles from Chennai is cherished gift.
- ¹⁵ It could be argued that lower life forms like ants and birds display only (genetically) scripted behaviour. Humans however, have a more cognitive approach in which they reason about their goals and means to achieve them.
- ¹⁶ We use the standard description of a rule. The “action” here is what the rule does, not to be confused with an “Action” in the story that an actor does.
- ¹⁷ Using a discriminatory network somewhat like the Rete Net (see Chapter 6).
- ¹⁸ We will not divulge the ending here. The interested reader is referred to Meehan’s publications.
- ¹⁹ Any large complex system is likely to have a flavour of both algorithms and vast amounts of diverse knowledge. For a definition of neats vs scruffies, see <http://www.computer-dictionary-online.org/neats%20vs.%20scruffies.htm?q=neats%20vs.%20scruffies>
- ²⁰ In particular, two negative edges do not make a positive inheritance. If Penguin \rightarrow Hexapoda and Hexapoda \rightarrow Mammals are edges in a graph, Penguin \rightarrow Mammal is not a valid path.
- ²¹ See http://en.wikipedia.org/wiki/Gottfried_Leibniz
- ²² In the mathematical notation used in some other literature, the concept [AND Person Female] is written as Person \sqcap Female (see Box 14.2).
- ²³ Alternatively, one could only add those children to P that subsume d . In either case, each child has to be tested for subsumption exactly once.
- ²⁴ One has the option of adding a node with the description d to the

taxonomy. This is a design choice one has to make. The question is whether one needs a minimum number of instances to create a new concept in the taxonomy.

²⁵ Dictionary.com, "ontology," in *The Free Online Dictionary of Computing*. Source location: Denis Howe. <http://dictionary.reference.com/browse/ontology>. Accessed: December 14, 2009.

²⁶ Philosophers treat Ontology as a proper noun concerned with *all* things that exist. The phrase "an ontology" is a variation introduced in computer science pertaining to the framework associated with a particular domain.

²⁷ <http://www.w3.org/TR/owl-guide/>,
http://www.w3schools.com/RDF/rdf_owl.asp

²⁸ <http://protege.stanford.edu/>

²⁹ In the literature, these are also called the *extent* and the *intent* respectively.

³⁰ See http://en.wikipedia.org/wiki/Formal_concept_analysis

³¹ Instead of $\{A, B, C, D, E, F\}$

³² Observe that by using linguistic terms, one avoided addressing the fact that both "think" and "believe" are used in the same sense here.

³³ A conceptual subgraph of a CG is a subgraph of the CG that is also a CG.

³⁴ http://en.wikipedia.org/wiki/Gestalt_psychology

³⁵ See also http://en.wikipedia.org/wiki/Formal_concept_analysis

Memory and Experience: Case Based Reasoning

Chapter 15

Humankind is a problem solving species. There are basically two approaches to problem solving. One is the first principles approach, in which the agent does *search* in a space generated by domain modelling. The second is a knowledge based approach, in which the agent knows somehow in advance what the solution is likely to be. And the two are not entirely disjoint. As we have seen in the early chapters, we try and improve the performance of search by knowledge encoded as heuristic functions. On the other hand, when the system is knowledge based, one has to employ search to find the applicable pieces of knowledge. In whatever form and role it occurs, knowledge has to come from somewhere. The form in which the knowledge is held and deployed by the agent may vary, but typically such knowledge is the end product of *experience*. This experience might be the agent's own experience, or may be a lesson learnt and passed on by someone else.

In this chapter, we explore building systems that store and reuse problem solving experiences. Very often, the problem solver has a *dynamic memory* which stores experiences in an *explicit form* to be used for future problem solving. We will refer to this approach as a *memory based* approach, in which the agent uses information stored in its memory in some form to solve a given problem.

Learning systems also capitalize on experience, but they use the experiences to build and refine an implicit representation of what is known as a *target function*. Neural networks and decision trees are examples of learning systems. Rule based systems can also be thought of distilling experience into modular *if-then* chunks of knowledge.

The first principles approach to problem solving too does not operate in a vacuum. It does need to represent knowledge of the domain, the relations between them, and the knowledge of effects of agent choices in the domain. We refer to such an approach as a *model based* approach, because the simulation is done over some model of the problem domain. One could say that the model based approach uses *deep knowledge* (Struss, 2008) because it involves the representation of a model of the domain, and then experiential or heuristic knowledge can be thought of as *shallow knowledge*. Figure 15.1 highlights the broad difference between the *model based* and *memory based* approaches.

The popular puzzle known as the Rubik's cube (Figure 15.2) is a well known problem that illustrates the use of two techniques, and also the efficacy of the knowledge based approach. A Rubik's cube is a cube with the six faces coloured with six colours. The cube is sliced in nine squares on each of its faces, and each face of nine cubelets can rotate around an axis perpendicular to the face. Repeated rotations can jumble up the colours of the cube, and the goal is to get the faces back into one colour again. In the seventies, when the Hungarian architect Erno Rubik invented the puzzle, it became a rage and many people could be found *trying* to solve it obsessively.

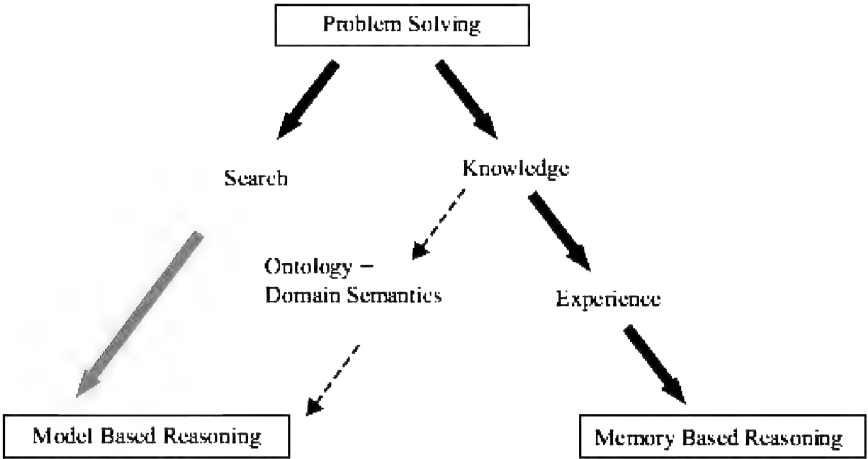


Figure 15.1 Model based and memory based reasoning.

The trial-and-error methods would fail more often than not, because the state space is huge, containing about 4.3×10^{19} states. More importantly, it has not been possible to construct a heuristic function that will monotonically reduce on the solution path. The intuition of assigning better values to states with more cubelets of the same colour together, soon leads to local maxima. Consequently, most human solvers use a peak-to-peak approach described in Chapter 3. In a matter of a few years, systematic solutions were found and learnt (memorized) by people, so that even young children *knew* how to solve it. Currently, the issue in human competitions is how fast you can solve it, and champions do it in a matter of seconds.

This example illustrates two contrasting advantages of the two approaches. When a solution or a solution method has been learnt, knowledge acts like a sword cutting a path through the combinatorially exploding number of possibilities. On the other hand, if a solution is not known in advance, only a first principle approach can find one, but the problem solver has to sift through a large number of combinations. Figure 15.3 depicts the role of knowledge in identifying a solution amongst the many possibilities that lie ahead. It depicts a solution or a plan being retrieved from the memory and used to synthesize future actions.

Knowledge, from this perspective, is the carry-forward from the past to the future. Memory is the seat of knowledge.

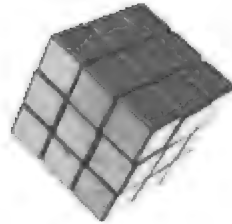


FIGURE 15.2 The Rubik's cube.

One can identify three different ontological forms of knowledge. The first is *direct experience* which one can symbolically store in the form of episodes. We can call each of these stored experiences a *case*. As many episodes are encountered over a period of time, two more compact forms of knowledge emerge by a process of abstraction and generalization. The first is *semantic* in nature, which we can call a *model*. A model is an abstraction of reality that can be used to simulate events in the real world. A model is an embodiment of “how things work”. The second is *operational* in nature, or heuristic knowledge, often represented as rules. Heuristic rules may suggest “what to do” in a situation, for example “if one is putting on weight, one should exercise more”; or what to expect in the domain, for example, “if the dog is barking, there may be someone about to knock on the door”. There can be other compact forms of knowledge as described in machine learning literature (Mitchell, 1997), for example decision trees, but they are variations on the theme. Figure 15.4 shows the relation between the three forms of knowledge, *cases*, *rules*, and *models*.

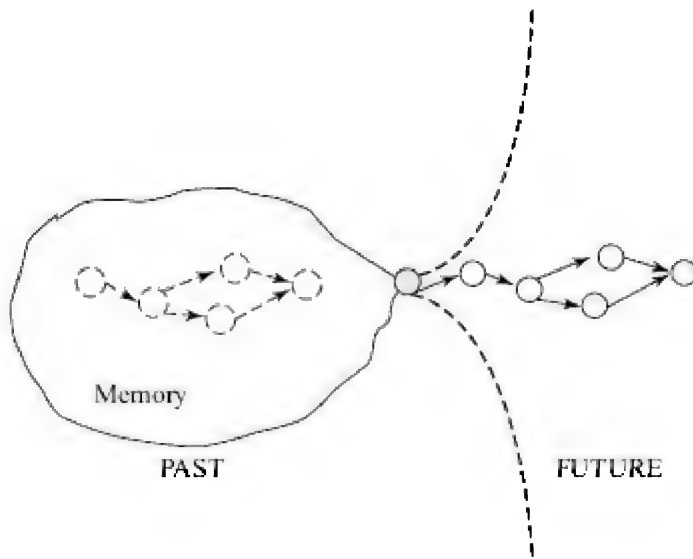


FIGURE 15.3 Knowledge is the carry-forward from the past to the future.

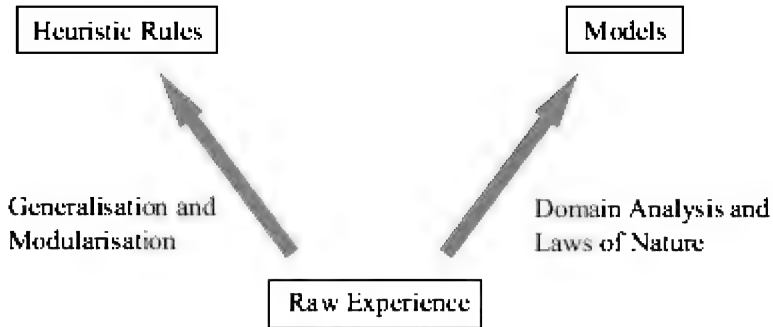


FIGURE 15.4 Models and rules are drawn from raw experience. A case based system organizes episodes of raw experience.

Knowledge can be a carry-forward from the past to the future when there are things that do not change from the past to the future. Then a sentient agent may observe something, remember it, and exploit it in the future.

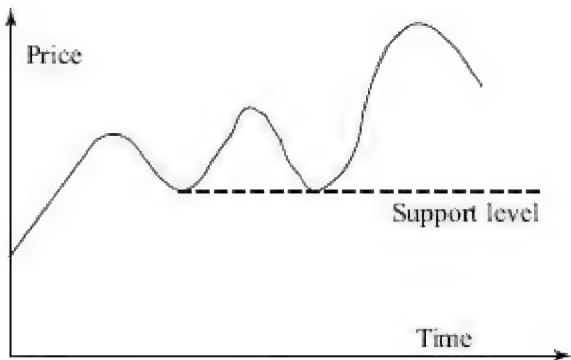


FIGURE 15.5 Notion of support is memory based.

Another interesting example comes from the world of stock trading. A section of the market analysts, known for some reason as “technical analysts”, study historical stock prices and make predictions about what is likely to happen. Two common terms they use are “support” and “resistance”. By support, one means a price below which the stock is not likely to fall, because many buyers will come in. Likewise by resistance, one means a price above which a stock has difficulty rising, because sellers come to the fore. Interestingly, these levels are determined by experience. Figure 15.5 illustrates how a simple notion of past experience of a stock price might determine a support level. In practice, analysts use more sophisticated measures like the 50- and 200-day moving averages, or their crossovers, or more complex patterns with names like “head and shoulders bottom” or the “candlesticks” patterns invented by the Japanese for rice trade in the 17th century (see for example (Murphy, 1999), (Nison,

2001)). Technical analysis is essentially forecasting, based on patterns in time-series data. This is not limited to the stock market, and can be applied anywhere a pattern in a time series data is predictive of something of interest. Thus, the technical analysis of stock prices is essentially memory based. The other approach is known as “fundamental analysis”, and consists of evaluating the profits and growth of a company. It is essentially a model based approach.

It may be observed that technical analysts are comfortable only when they are in familiar territory, and are unable to make predictions when a stock (price) breaks into uncharted territory. This is a characteristic of experience. Experiential knowledge, in general, is unable to deal with situations that have changed beyond recognition. A poignant example of that is the inability or the unwillingness of humankind to accept the dangers of climate change, because we have not experienced it in recorded history, and model based answers are imprecise and open to debate. Another distressing example, where we did pay the cost of missing knowledge, was when the Tsunami devastated the coastlines of southern Asia in December 2004. The previous recorded case of a Tsunami in India was in 1941, too far back in time to exist in the minds of people when it struck again. Japanese folklore, on the other hand, is replete with tales of the Tsunami, and a child there would have understood the import of the unusual receding of the sea waters.

Memory, knowledge and experience are not just the *forte* of individuals but also of societies and organizations. The emergence of language has made it possible for societies to share and pass on knowledge. Often such knowledge is part of folklore and gets distilled in the form of customs and rituals, which may survive long after the reasoning behind them is remembered.

15.1 Case Based Reasoning

Human memory is complex. We have the ability to absorb information continuously since our birth. We constantly generalize from our experiences, and yet are able to recall individual episodes from our memories. Very often, these are buried deep inside somewhere, and are remembered by a process of reminding that is still not well understood. Cognitive scientists talk of long term memory and short term memories. The notion of organizing information in knowledge structures has long been a subject matter of interest. Much of this work has been done from the perspective of perception and understanding, but is also applicable to planning and problem solving. Various terms have been used for the process of aggregating information. The most commonly used term is *schemata* (Sowa, 1984). Herbert Simon used the term *chunks* (Simon, 1974) and postulated that human expertise involves a larger and larger collection of such chunked patterns. He estimated that good chess players have about a 1000 of them, while masters could have a hundred times more. Marvin Minsky introduced the notion of *frames* (Minsky, 1975), a

structure used to aggregate information and a precursor of *objects* (see Chapter 14).

A considerable amount of work done at Yale University in the seventies introduced knowledge structures like Conceptual Dependency, Scripts and Memory Organization Packets with focus on natural language understanding (Schank and Abelson, 1977), (Schank and Riesbeck, 1981), (Schank, 1982; 1999). We looked at some of these concepts in Chapters 13 and 14. From this body of work emerged the notion of *Case Based Reasoning* (CBR) as a methodology of problem solving (Riesbeck and Schank, 1989), (Kolodner, 1993). The simple idea behind CBR is that a problem solver should remember what works and what does not work. It solves problems by consulting its repository of cases in its memory. If a new solution is devised, by any means, then it is added to the repository. This has been summed up succinctly as a process of four *R*'s (Aamodt and Plaza, 1994) and is illustrated in the Figure 15.6 adapted from their work.

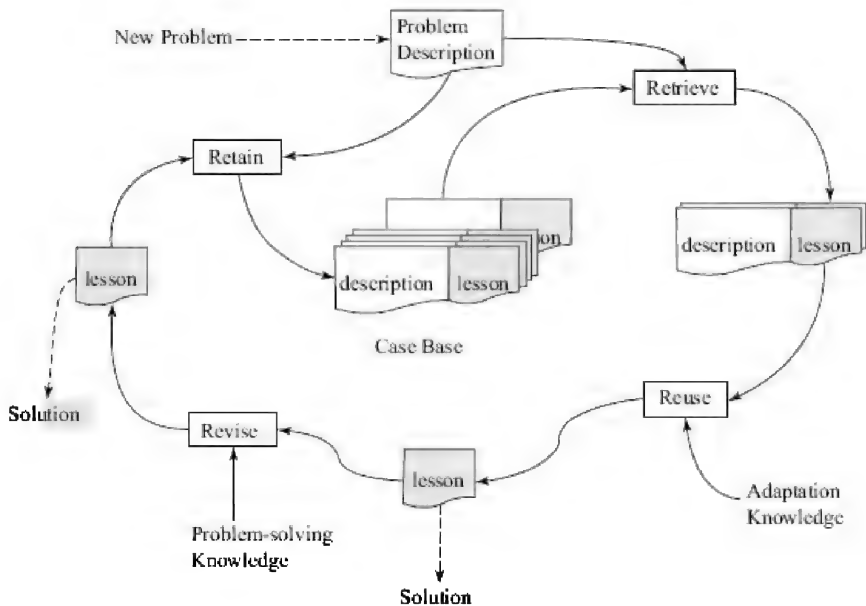


FIGURE 15.6 The case based reasoning cycle.

A case is made up of two constituents, a *description* and a *lesson*. The first part contains a representation of the problem description that is used to access cases, and the second contains the key to constructing the solution. The four *R*'s define the CBR methodology as follows:

- Given a new problem to solve, *retrieve* the best matching case or set of cases from the case base.
- Try and *reuse* the lesson stored in the retrieved case(s) to construction by *adapting* the retrieved lesson(s).
- If necessary, bring in more knowledge to *revise* the solution to suit the current problem. This knowledge may come from a different

problem solving system or from a human expert.

- *Retain* the new revised solution, by adding it to the case base with appropriate indices for retrieval¹.

The CBR system thus implements a simple form of *dynamic memory* that learns by adding new cases, as and when new problems are encountered and solved.

Conceptually, we can describe the terms used in CBR as follows. Let p be a problem from some domain. Let d be a description of the problem constructed using some vocabulary. Let l be a lesson expressed using some vocabulary. The vocabulary used for expressing lessons may be the same as the one used for descriptions, or it could be different. A case is a description-lesson pair $\langle d, l \rangle$. The *utility* $u(p, l)$ of a lesson l for a problem p is a value between 0 and 1, and signifies how good the lesson l is for solving problem p .

Given a problem to solve, one would ideally like to retrieve cases that have the highest utility amongst the cases in the case base. However, utility is difficult to define and measure. Utility could depend upon the ease with which a solution is constructed, or upon the quality of the solution found, and it could also depend upon the ease of implementing the solution. In any case, utility can only be determined after the solution constructed by the lesson is *applied* to the problem, which of course defeats the purpose of *finding* a good case to construct the solution. Thus, while we *are* interested in cases with high utility lessons, we *have* to use other criteria for finding them.

The utility value of a lesson becomes known only in the future, after the lesson is applied to the problem. A good case is one which has a lesson of high utility for the current problem. A knowledge-based approach retrieves a (good) case by inspecting its past, stored in the case base. The criterion for choosing the case is *similarity*. Case based reasoning is based on the following premise:

Similar problems have similar solutions.

Also, the problems we encounter *are* often similar. This is based on the fact that things change continuously and smoothly in the world, and that there are situations that often repeat themselves. The best case is then that whose description is most similar to the current problem description. To measure this, we use a similarity function $sim(d_i, d_j)$ that returns a value between 0 and 1, where 1 stands of identity or total similarity, and 0 represents no similarity or total dissimilarity.

Thus, case based reasoning works as follows,

- Given a problem p , construct a description d of the problem.
- Retrieve from the case base a case $c_j = \langle d_j, l_j \rangle$, such that $sim(d, d_j)$ is highest amongst all cases in the case base (retrieve).
- Adapt the retrieved lesson l_j to construct a solution for p (reuse).
- If the solution has low utility, construct a new solution and a lesson l (revise).
- When a new problem solving experience occurs, add a lesson l to

form the case $c = \langle d, l \rangle$ and add it to the case base CB (retain).

In practice, one may use more than one similar case to construct the solution. We often refer to this process of retrieving K most similar cases as K Nearest Neighbour (KNN) retrieval. When CBR is used for classification then the solution could be the majority class label amongst the K retrieved cases. When solving a planning problem, a plan may be *composed* from components extracted from different cases in the retrieval set. For solving a troubleshooting (helpdesk) problem, one may use statistical information of problems encountered in the past to suggest the most likely solution from the retrieved set.

One may also extend the case representation to include an *Outcome* field in addition to the Description and the Lesson. This contains the result of applying the lesson to the problem, and could be used to remind the user that a particular lesson did not work. In a way, it is like storing the utility of the case, and one would expect to use only high utility cases. This would be particularly useful in domains where the outcomes are stochastic, for example in foundrylike situations when the same manufacturing process sometimes yields defective products. One can then use statistical information from past usage to decide how likely the case is to be successful (see for example (Selvamani and Khemani, 2003)).

To implement the CBR methodology, one has to address the following issues,

1. How are cases represented? This applies to both the description part and the lesson part.
2. How is the similarity between two descriptions computed?
3. How does one retrieve the desired cases efficiently?
4. How is the solution constructed from the lesson?

An important issue is the problem description. The problem description is the one that defines the relevance of a case. One needs to ensure the *utility distinguishability* of the representation (Bergmann, 2002). This means that if the descriptions in two cases are identical then the corresponding lessons will have the same utility for any given problem. Equivalently, if the lessons in two cases have different utility values for a given problem then the two descriptions in the case must be different.

$$\forall p \forall c_i \forall c_k (c_i = \langle d_i, l_i \rangle \wedge c_k = \langle d_k, l_k \rangle \wedge d_i = d_k \supset u(p, l_i) = u(p, l_k))$$

or,

$$\forall p \forall c_i \forall c_k (c_i = \langle d_i, l_i \rangle \wedge c_k = \langle d_k, l_k \rangle \wedge u(p, l_i) \neq u(p, l_k) \supset d_i \neq d_k)$$

Apart from these, there are issues of maintenance as well. One has to decide which cases to retain, which ones to delete, and what new cases are to be added. Cases cannot be added in an unrestricted manner because the retrieval performance goes down as the case base grows in size. In a dynamic environment, some cases may become obsolete and may need to be retired.

15.1.1 The Retrieval Task

Given a case base and given a problem to solve, the description of the problem is matched with the descriptions of the cases in the case base. The set of retrieved cases constitute the retrieval set R . The retrieval goals may be one of the following,

All Best The retrieval set contains all cases that have maximal similarity with the problem description.

That is,

$$R \leftarrow \{C_i = (d_i, l_i) \in CB \mid \exists C_p = (d_p, l_p) \in CB \text{ and } sim(d, d_p) > sim(d, d_i)\}$$

This goal says that the retrieval set must contain all the cases with the highest similarity. Observe, that this does not mean that all the retrieved cases are identical, but only that their similarity with the problem description is the same.

K Nearest Neighbours The retrieval set must contain the K most similar cases, or the K nearest neighbours when we view the cases in some space.

$$R \leftarrow \{C_i = (d_i, l_i) \in CB \mid \exists C_p = (d_p, l_p) \in CB \setminus R \text{ and } sim(d, d_p) > sim(d, d_i) \text{ and } |R| = k\}$$

K Nearest Neighbours retrieval or KNN retrieval is the most commonly used criterion. Observe that the All Best criterion could retrieve just one case. Very often, one requires more than one case. This is because CBR is often applied to ill understood problems, and only the best *matching* case may not provide the best solution. In practice, with a retrieval set of size K , the solution may be constructed from the K lessons, either by majority voting or by similarity weighted averages. Such an approach also guards against a noisy case base, where a few cases may be erroneous.

Threshold Similarity The retrieval set contains all those cases that have a similarity above a chosen threshold t .

$$R \leftarrow \{C_i = (d_i, l_i) \in CB \mid sim(d, d_i) > t\}$$

This could be used when one wants to ensure that only cases with a desired similarity or better are retrieved. Note that the retrieval set may be empty.

15.1.2 Diversity

Cases in the retrieval set that are maximally similar to the query are also likely to be similar to each other. We call the retrieval set obtained by the criterion given above as the Standard Retrieval Set (SRS). In some

applications, one may want the cases in the retrieval set to be different from *each other*, and the retrieval set to be as *diverse* as possible (Smyth and McClave, 2001). Diversity is important in *recommender systems* where a user is looking for a product and may also want some choice. For example, if one is consulting a web based real estate system looking for a home, an entire new apartment complex may match the stated requirements. Then, the SRS would probably contain all apartments from the same complex, which may not offer enough choice to the user. Diversity of a retrieval set $R = \{c_1 = \langle d_1, l_1 \rangle, c_2 = \langle d_2, l_2 \rangle, \dots, c_k = \langle d_k, l_k \rangle\}$ is defined as,

$$\text{Diversity}(R) = \frac{\sum_{i=1..k} \sum_{j=1..k} (1 - \text{sim}(d_i, d_j))}{\frac{n}{2} \times (n - 1)}$$

While choosing a case c to add to a partially constructed retrieval set R , one can also consider how different it is from the cases already in the retrieval set. The relative diversity of a case c w.r.t. a set R is defined as,

$$\begin{aligned} \text{RelDiversity}(c, R) &= 1 \text{ if } R = \{ \} \\ &= \frac{\sum_{i=1..m} (1 - \text{sim}(d_i, d_i))}{m}; \text{ otherwise} \end{aligned}$$

The case to be incrementally added to the retrieval set would be one that maximizes a combination of similarity with the query and relative diversity with the retrieval set. One definition of the combined score, called Quality, is,

$$\text{Quality}(c, R) = \alpha * \text{sim}(d, d_{\text{Query}}) + (1 - \alpha) * \text{RelDiversity}(c, R)$$

where d_{Query} is the description of the given problem or query and α is a parameter used to control the importance of the two components in the final score. While similarity is a function of the query and a candidate case, relative diversity depends upon cases added to the retrieval set so far.

15.1.3 Forms of CBR

The basic process in CBR is that a user describes a problem and the system comes up with a solution. The system does this by retrieving the relevant case from its memory and constructing a solution from the lesson stored in the case. The process of retrieval and reuse is determined by the underlying representation. Three basic schemes have been identified (Lenz et al., 1998), (Bergmann, 2002).

1. Conversational CBR This focuses on the conversation that may take place in a helpdesk situation, in a Customer Relationship Management

(CRM) system. The questions that an agent may ask the customer assume the greatest importance and find a place in the representation system.

2. Structural CBR Cases are described using a *well defined vocabulary* of attributes and their values. The attributes may be organized on the relations between them.

3. Textual CBR Very often, organizations keep records of the problems they encounter and solve as jottings in natural language. When cases are represented in free text, some text processing techniques may have to be applied.

In the following sections, we explore each of the three avatars of CBR and look at issues of *representation*, *similarity* and *retrieval*.

15.1.4 Conversational CBR

Conversational CBR derives its name from the focus on conversation between an agent and a user. The agent may be a helpdesk operator in a diagnosis or troubleshooting situation, while the user could be a help seeker. Given the increasing propensity for customer support, a CBR system can come in handy to support a helpdesk operator engage in a meaningful conversation with the user. Some of the earliest successful applications of CBR were in helpdesk situations (for example, the Compaq's *Smart* system (Acorn and Walden, 1992)). More recently, conversational CBR has also found application in *recommender systems*. Recommender systems are electronic commerce applications designed to help a customer choose an appropriate product.

Instead of taking the requirement (or query) in one go, it is done incrementally via a series of questions. In the early systems, the questions were chosen by the agent, and used to sift through the possibilities. More recently however, there has also been work on *mixed initiative* systems in which both sides can choose what question to address next (see for example² (McSherry, 2002a)). This gives the user an opportunity to focus on the features she is interested in. For example, if one is looking to rent a flat, the system might start asking questions in a particular order, but you might first bring up the need for a balcony.

While conversational CBR can be done with any kind of representation, the name was initially used by a representation that focused on the questions. A case in conversational CBR is a collection of questions and answers, along with a diagnosis and actions to be taken. In addition, there may be a case number and a title. The following is an example of a case in conversational CBR³.

Title: PC not booting

- Q1. Do you hear a BIOS beep code when you turn on the PC? Yes.**
- Q2. Do you know if your BIOS is made by American Megatrends? Yes.**
- Q3. Did you hear a beep before the BIOS startup screen is displayed? Yes.**
- Q4. Did the boot process continue? No.**
- Q5. Did you see the BIOS startup screen displayed? No.**

Problem: Dynamic RAM(DRAM) Refresh Failure.

- Action:**
- 1. Troubleshoot your motherboard**
 - 2. Treat this as a memory failure.**

FIGURE 15.7 An example of a conversational case.

The cases are arranged in a tree structure that guides the search for the relevant case by questions posed at each internal node, as illustrated in Figure 15.8. In the figure, nodes are labelled with distinct question labels, but they do not have to be all distinct. For example, Q_{31} and Q_{34} could be identical.

Retrieval happens by the system asking a sequence of questions, starting at the root, and choosing the edge to traverse, based on the answer given by the user. At the end of the path is a leaf node containing the case, which contains the diagnosis and the therapy actions for the problem indexed by the questions and their answers. The high level retrieval algorithm is given in Figure 15.9.

Conversational CBR systems can be built rapidly where the developer can pose the questions that make up the path for each case. No domain representation is needed if the tree is constructed manually. All one needs to do is decide on what questions will index the case. At least one of the questions in a new case must be a node in the existing tree, with a different answer. The path to the new case will divert from this node, and the new questions may be added below that. Figure 15.10 outlines an algorithm to add a new case and its associated conversation to the case base.

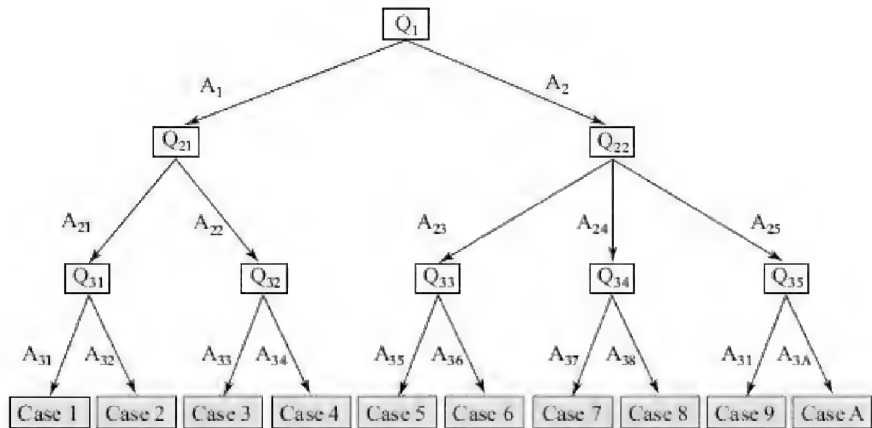


FIGURE 15.8 In conversational CBR, cases are stored as the leaves of a tree whose internal nodes are questions. Retrieval involves traversal from the root to a leaf navigated by the answers to the questions provided by the user.

```

ConversationalRetrieval(tree)
1  n ← Root(tree)
2  while not LeafNode(n)
3    do Ask Question Q in Node n
4    Let A be the set of answers
5    Accept response R from user, including "don't know"
6    if R ∉ A
7      then return set of leaves below Node n
8      else Let node m = child(n) such that edge(n,m) is
                                                    labeled with R
9          n ← m
10 return n

```

FIGURE 15.9 The retrieval in conversational CBR is done by traversing the indexing tree, based on user responses. If the user fails to give a response that labels an edge to a child, the algorithm returns the leaves under the subtree, rooted at the node containing that question.

If the answers terminate at an internal node then all the cases in the subtree could be retrieved. But the order in which the questions are posed is fixed, and for any question to be asked, all previous questions must have been ordered.

The above algorithm can be used to construct the case base from scratch, adding cases one by one. In conversational CBR, this task is carried out by a human who knows questions and answers used. In order to automate the process, one needs to have an explicit case representation that can be used to generate answers to the relevant questions. When the above algorithm reaches a leaf node, one has to devise a new question in addition, to discriminate between the case stored there and the new case. This is only possible when the case representation contains some domain specific information. We will look at the process of automatically constructing indexing trees for structural CBR later in this chapter and also in the chapter on Machine Learning.

```

AddConversation(case, tree)
1  n ← Root(tree)
2  while not LeafNode(n)
3      do Let Q be the question at node n
4         Let A be the set of answers
5         Let R be the answer to Q in case, including "don't know"
6         if R ∉ A
7             then Create a new child c of n with answer R for its edge
8                 Attach case to c
9                 return
10        else Let node m = child(n) such that edge(n,m) is labeled with R
11            n ← m
12    /* reached a leaf containing a case cold */
13    Find some difference D between cold and case
14    Construct a question Q to address the difference D
15    Store Q in node n
16    if cold answers Q with A1
17        then Create child n1 with edge label A1 and attach cold to n1
18    if case answers Q with A2
19        then Create child n2 with edge label A2 and attach case to n2

```

FIGURE 15.10 A new case may be inserted in the case by starting at the root and answering the question stored there. The case follows the path where it has appropriate answers. When it does not have an answer, a new arc is created to accommodate the new case. If there are no more questions in the path, a new question is added.

Given that there is no domain modelling in conversational CBR, there is also no notion of similarity. One could introduce a notion of similarity based on the location of two cases based on their location in the tree (see the section on structural CBR), but that is not sound because different trees can be constructed for the same cases. Furthermore, similarity is not used for retrieval.

The greatest difficulty with this approach is that trees have to be handcrafted. In addition, maintenance is also cumbersome since the tree has to be edited manually. Conversational CBR would be useful when there is high traffic of consulting in a domain with small trees that can be handcrafted. For more complex domains, some explicit modelling of the domain could facilitate more flexible retrieval.

In the following section, we introduce structural representation, based on a well defined vocabulary of attributes and values. When this is done, we will see that it will be possible to construct trees for indexing automatically. We will still be able to implement retrieval by an incremental dialogue, like in conversational CBR. In addition, the questions to be asked could be chosen dynamically. We will also be able to allow the user to select questions in a mixed initiative mode.

15.1.5 Structural CBR

In structural CBR, a case is defined using a well defined vocabulary of attributes.

$$V_{att} = \{A_1, A_2, \dots, A_n\}$$

In practice, we tend to use attribute names, like *height*, *weight*, *cost*,

that are meaningful to us. Associated with each attribute A_i is a domain D_i of values that can be assigned to the attribute.

$$D_i = \{v_{i1}, v_{i2}, \dots, v_{ip}\}$$

The domain need not be finite or even countable. Associated with each attribute A_i is a type T_i that defines the type of the values in the domain D_i . The commonly used types are Boolean, Integer, Real, Symbolic and Text. The Symbolic types may be further divided into Ordered, Unordered and Taxonomic types.

The lesson part of the case may or may not have a different vocabulary. In classification problems, the lesson may and attribute with a class label. In troubleshooting problems, the lesson may be a diagnosis and a set of actions to be performed. In recommender systems, there may be no distinction between the description and the lesson attributes. The user may partially describe a product, and the solution may be a full description of the best matching product. Here, any of the attributes could be the problem description, and the unstated attributes, the lesson used for *case completion* (Burkhard, 1998), (Hayes et al., 2001). One can think of the last example as an implementation of a *content addressable memory* (see (Bechtel and Abrahamsen, 2002)), where objects are retrieved by describing them partially.

Cases are made up of a collection of attributes and values. Similarities between cases are defined by aggregation of similarities at the attribute level. The simplest representation is when the case is simply a set of attributes with their values. These are often referred to as *flat* cases. More complex cases may have the attributes organized by relations between attributes and collections of attributes in framelike structures. We will look at these *structured* cases later. The techniques for dealing with flat cases can easily extend to structured cases.

The common feature in structural CBR is that similarity between cases, or a case and a query, is computed by aggregating the similarities at the attribute level. The similarities at the attribute level are called *local* similarities, and the similarity at the case level is called *global* similarity. We first investigate local similarities and then look at how they are aggregated to determine global similarity.

15.1.6 Local Similarity

Let A_1 and A_2 be two attributes with the same domain D . One may belong to the query, and the other correspondingly to a case. Let V_1 and V_2 be the two values from D assigned to A_1 and A_2 respectively. The question then is how similar the two values V_1 and V_2 are. We need to define a similarity function sim_D such that,

$$sim_D(V_1, V_2) = 1 \text{ if the two values are identical or totally similar}$$

$$sim_D(V_1, V_2) = 0 \text{ if the two values are totally dissimilar}$$

and with other values in the range $[0, 1]$, such that higher values reflect higher similarity.

The answer depends upon the *type* of the domain D .

It may also depend upon *what* the values represent. If for example the values come from a physical system in which the values can be represented in *qualitative intervals* then within an interval, one would have high similarity and across intervals low similarity. For example, we may think of values of temperature of water between room temperature and boiling point as one interval. Then 90° Celsius would be more similar to 70° Celsius than to 102° Celsius because the latter is outside the interval and water goes into a different state. Likewise, if someone has a nine-to-five work schedule then 1630 hours will be more like 1430 hours than 1730 hours.

In the absence of such knowledge, we rely upon the type T associated with the attribute. The following are the more commonly used types and their similarity functions.

Boolean Attributes

When the type of the attribute is Boolean then the similarity is also a yes/no option. Either they are the same or they are different.

$$\begin{aligned} \text{sim}_{\text{Boolean}}(V_1, V_2) &= 1 && \text{if } V_1 = V_2 \\ &= 0 && \text{otherwise} \end{aligned}$$

Numeric Attributes

When the values V_1 and V_2 come from an integer or real domain then similarity depends upon the *difference* or the *distance* between the two values. But we *also* need to know the range from which the values come.

Consider for example a difference of 3 for numeric values. That is,

$$V_1 - V_2 = 3 \text{ or } V_2 - V_1 = 3$$

How similar are the two values? If the values were $V_1 = 2$ and $V_2 = 5$, you might consider them to be less similar than if they were 65 and 68. This is because at a subconscious level, one associates a background range with these numbers. For example, 2 and 5 might be the (very different) weights in kilograms of newborn babies while 65 and 68 might be the (quite similar) weights of two adults. The *range* of baby weights is much smaller than the range of weights of adults.⁴

Thus, local similarity between two numeric values V_1 and V_2 can be defined as,

$$\text{sim}_{\text{Numeric}}(V_1, V_2) = 1 - (|V_1 - V_2|) / (V_{\max} - V_{\min})$$

where V_{\max} is the maximum value in the domain D and V_{\min} the minimum. By definition,

$$\text{sim}_{\text{Numeric}}(V_{\text{max}}, V_{\text{min}}) = 0$$

The expression $|V_1 - V_2|$ is the absolute value of the difference. This implies that the similarity is symmetric, and does not depend upon which one of the values is greater. There can be situations when one may want to define a similarity function that is not symmetric. Consider the task of recruiting pilots. Then the case may have a value V_1 for height that is acceptable for the job. Let the query have value V_2 representing the height of an aspiring candidate. In such a situation, if V_2 is greater than V_1 then one may want to assign a greater similarity value than if it were the other way round, assuming that one of the requirements for the job is that of a minimum height. Likewise, if a query states a desired price for a product then a lower price tag on a product may match better than a price tag higher by the same amount.

String Attributes

String type attributes have a value that is a sequence of characters from some alphabet. The alphabet could be $\{0, 1\}$ for binary codes, the set $\{C, A, G, T\}$ for chromosomes, or the set of letters $\{a-z\}$ for text attributes.

A string may be treated as a symbol name for which a specific similarity function may apply. For example, the strings “blue” or “Wednesday” may be treated as symbols. A string may be treated as a piece of text, which may have its own similarity function. We will look at these options later.

Otherwise, the similarity between two strings may be defined as a function of some notion of difference or distance defined on strings.

Levenshtein Distance One measure of difference in strings is the notion of edit distance, which counts the number of changes one has to make to convert one string into another (Levenshtein, 1966). For example, given the two strings,

S_1 : brick and mortar

S_2 : click and portal

one can be changed into the other by changing four letters in either string. In addition to substituting characters, one may also allow deletion and insertion of characters. This will allow us to compare strings of different lengths. For example, given the strings,

S_3 : nuclear energy is safe

S_4 : nuclear energy is unsafe

S_3 can be converted into S_4 by inserting the characters “un” in S_4 . However, as discussed in Chapter 5, the number of ways a string of length n can be aligned with a string of length m , when insertions and deletions is allowed, is the number of distinct paths in an $n + 1$ by $m + 1$ rectangular grid, with diagonals from left to right as well. The number of such paths is

$P(m, n)$, which can be prohibitively large as the strings become longer. Consequently, a simpler similarity function is desirable.

Hamming Distance One simple form of edit distance is the Hamming distance. This distance measure counts the number of locations where the two strings differ, and is defined for strings of equal length (Hamming, 1950). The distance between strings S_1 and S_2 is four, while it is not defined for strings S_3 and S_4 . One could in principle pad the shorter string with blanks, but the resulting measure of distance may not be intuitively acceptable. Hamming distance is also a *metric*. That is, it satisfies the properties⁵,

1. $d(X, Y) \geq 0$ (non-negativity)
2. $d(X, Y) = 0$ if and only if $X = Y$ (identity of indiscernibles)⁶
3. $d(X, Y) = d(Y, X)$ (symmetry)
4. $d(X, Z) \leq d(X, Y) + d(Y, Z)$ (triangle inequality).

The maximum possible Hamming distance between two strings of length n , is n . Let h be the hamming distance between two strings. Then, the normalized hamming distance h' is h/n . Similarity between the two strings X and Y could then be defined as,

$$\begin{aligned} \text{sim}_H(X, Y) &= 1 - h' \\ &= 1 - h/n \end{aligned}$$

Using this definition, the similarity between strings S_1 and S_2 is,

$$\begin{aligned} \text{sim}_H(S_1, S_2) &= 1 - 4/16 \\ &= 0.75 \end{aligned}$$

n -gram Similarity When we compare strings by mapping characters of one to the other, the outcome is critically dependent on the alignment. Further, insertion of even one character in the string, shifts the alignment of the entire remaining string.

When strings are natural language word sequences, this can lead to similarity measures that are susceptible to large fluctuations with small changes. When an attribute contains text, the similarity between two strings should ideally be compared, based on the *meaning* of the text. For example, the two strings “I feel like eating something” and “I am hungry” represent a similar state of a person, and should be treated to be highly similar. But that requires deep semantic knowledge. We will look at other approaches to matching text documents in Chapter 16. Here we look at a simple approach of making text matching more robust. This is based on representing the string as a collection of n -grams, where each n -gram is a sequence of n characters that occurs consecutively in the string.

Let the two strings X and Y be represented by sets of n -grams N_X and N_Y .

$$N_X = \{X_1, X_2, \dots, X_n\}$$

$$N_Y = \{Y_1, Y_2, \dots, Y_m\}$$

Then, the n -gram similarity between X and Y is given by,

$$\text{sim}(X, Y) = \frac{|N_x \cap N_y|}{\max(|N_x|, |N_y|)}$$

The set representation does not take into account repeated occurrences of n -grams. We can extend this by treating the n -gram representation as bags, treating each occurrence as distinct. The set of common n -grams will have to take this into account. Observe that the set representation ignores the *position* in which n -grams occur. This is similar to the vector space representation for documents made up of terms that we will investigate in the language processing chapter. One difference is that there is some positional information captured in the individual n -grams. The two sentences “John loves Mary” and “Mary loves John” will in fact have slightly different sets of trigrams, whereas treated as a bag of words in the vector space representation, the two become identical.

We illustrate the n -gram matching with the two strings S_1 and S_2 described above. We take n to be 3 and let T_{S_1} and T_{S_2} be the two sets of *trigrams* for the corresponding strings S_1 and S_2 .

T_{S_1} : {bri, ric, ick, ck_, k_a, _an, and, nd_, d_m, _mo, mor, ort, rta, tar}

T_{S_2} : {cli, lic, ick, ck_, k_a, _an, and, nd_, d_p, _po, por, ort, rta, tal}

$$T_{S_1} \cap T_{S_2} = \{\text{ick, ck_}, \text{k_a, _an, and, nd_}, \text{ort, rta}\}$$

$$|T_{S_1}| = 14, |T_{S_2}| = 14, |T_{S_1} \cap T_{S_2}| = 8$$

$$\text{sim}_T(S_1, S_2) = 8/14 = 0.57$$

The reader would have noticed that the trigram similarity between S_1 and S_2 is lower than the similarity based on the Hamming distance. This is because the two strings have many common characters in the same place. If this was not so, then the trigram method would have scored better. Consider a *slight* modification of the two strings with brick replaced by bricks,

S'_1 : bricks and mortar

S_2 : click and portal

Then, the set of trigrams for the first sentence is,

$T_{S'1}$: {bri, ric, ick, cks, ks_, s_a, _an, and, nd_, d_m, _mo, mor, ort, rta, tar} $|T_{S'1}| = 15$

The common trigrams are,

$$T_{S'1} \cap T_{S2} = \{\text{ick, _an, and, nd_, ort, rta}\}$$

$$|T_{S'1} \cap T_{S2}| = 6$$

The similarity now is,

$$\text{sim}_T(S'_1, S_2) = 6/15 = 0.40$$

The similarity has gone down, but is still a significant value. The Hamming distance does not apply because the two strings are of equal length. If we were to pad the second string with a blank at the end to make them equal length, the Hamming distance between the two strings would now be 14, and with string length going up to 17 the similarity is,

$$\text{sim}_H(S'_1, S_2) = 1 - 14/17 = 0.176$$

As one can see, the Hamming distance based similarity has plunged with the addition of the extra letter. This is because the inserted letter has disrupted the alignment between the two strings that was high initially.

The trigram based similarity, on the other hand, is much more robust.

We have used the common trigrams as a measure of similarity. We can also treat each trigram as a word or term of text, and use the cosine similarity for text described in the language processing chapter.

Symbol Attributes

When a string stands for a symbol then it must be treated as an atomic element. Symbol type attributes occur in many domains. We can treat names of colours like “red”, “orange” and “blue” as symbols. Other examples are the names of days of a week, months in a year, the codes assigned to products like cameras, printers, etc.

Symbol type attributes can be further classified into three types: *ordered*, *unordered* and *taxonomic*.

Ordered Symbols

Ordered symbol types may derive a notion of similarity from the nearness (or distance) between two symbols in the order. For example, the month of May could be considered more similar to April than to October. Similarity for ordered symbol types can then be computed, based on the index of the values in the order. Let A_O be an ordered symbol type attribute that can take values from the ordered set as V_{AO} given below,

$$V_{AO} = (V_1, V_2, \dots, V_T)$$

The similarity between two values V_i and V_k is given by

$$sim_{Ordered}(V_i, V_k) = 1 - (|k - i|) / (T - 1)$$

Unordered Symbols

Unordered symbols types may be treated in two ways. The simpler approach is to rely on equality. That is,

$$sim_{Unordered}(V_i, V_k) = \begin{cases} 1 & \text{iff } V_i = V_k \\ 0 & \text{otherwise.} \end{cases}$$

However, in some domains, the symbols may stand for some things that may have some inherent similarity. This is like saying that the symbols have some meaning (semantics) and similarity can be computed at the meaning level. Observe that this is what we said about words in natural language too. This is not surprising that our natural languages are essentially symbol systems, where words symbolize something.⁷

One can allow a user to define similarity between unordered symbol types. For example, if the attribute *Drinks* as part of a case structure for planning children's parties may have the values as follows,

$$V_{Drinks} = \{\text{Coke, Pepsi, Lemonade, Orange juice}\}$$

then a user could construct a similarity table explicitly, for instance, as follows,

Table 15.1 A similarity table for drinks

sim_{Drinks}	<i>Coke</i>	<i>Pepsi</i>	<i>Lemonade</i>	<i>Orange juice</i>
Coke	1	0.9	0.4	0.2
Pepsi	0.9	1	0.4	0.2
Lemonade	0.4	0.4	1	0.8
Orange juice	0.2	0.2	0.8	1

Other symbolic attributes, like colour, may have an underlying physical basis. Colours can be described by the frequency of light, but that would apply only to pure colours. Instead, one of the ways we describe a colour is as a point in the RGB space which is formed by taking three colours as primitives, and others can be described as combinations of the three "primary" colours. We can then treat the three RGB values as numeric attributes and compute their local similarities, and combine the three local similarities using some aggregation function.

Taxonomic Symbols

Taxonomies are partial orders in which one symbol is a *hyponym* of another. For example the symbol (or word) "car" is a hyponym for the symbol "vehicle", and "chair" is a hyponym of "furniture". The most commonly known taxonomies are in the animal kingdom, where life forms

are categorized into species and families.

The tree structure of a taxonomy gives a basis for defining similarity between symbol types. Consider, for example, a taxonomy of meal types shown in Figure 15.11. It could be an attribute you have to fix for a dinner party, or the menu card of an eclectic restaurant.

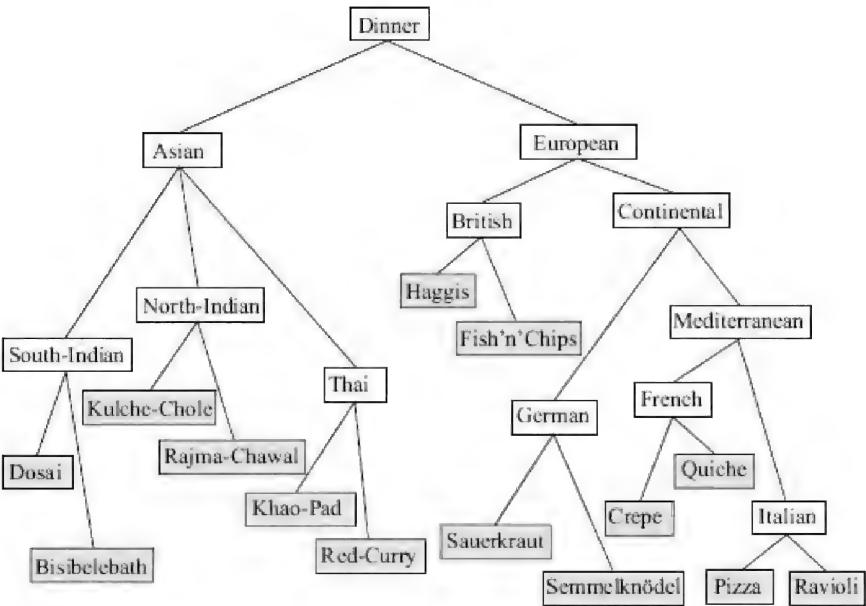


FIGURE 15.11 The dinner offered by a restaurant may have values that can be organized in a taxonomy.

Observe that only the leaves in the taxonomy represent specific food items. The internal nodes stand for collections of food items. But any node could be a value of the attribute. For example, you friend may choose “Thai” for dinner, meaning that any of the leaves below that node are acceptable. Or a restaurant may say that they serve “Continental” food, meaning that they serve all the items that are leaves below this node.

How does one define similarity between different values represented by the nodes in the taxonomy of values? The topology suggests that “Red-Curry” is more similar to “Rajma-Chawal” than it is to “Pizza”. Or one could say that it is closer to “North-India” than to “British”. Or one might say that “Thai” is more similar to “South-Indian” than it is to “Mediterranean”. All these comparisons can be handled as follows.

For two nodes, N_i and N_k , let $N_i \leq N_k$ denote that N_i is a descendant (or hyponym) of N_k , and let $\langle N_i, N_k \rangle$ denote the lowest common ancestor(LCA) of N_i and N_k . A node L is the lowest common ancestor to two nodes N and M iff $N \leq L$ and $M \leq L$ and there is no node L' such that $N \leq L'$ and $M \leq L'$ and $L' < L$. For example, “Asian” is the LCA of “Red-Curry” and “Rajma-Chawal”.

Then, a query node Q is more similar to a node N_i than to node N_k iff

the LCA of Q and N_i is a descendant of the LCA of Q and N_k . That is,

$$\text{sim}(Q, N_i) > \text{sim}(Q, N_k) \equiv \langle Q, N_i \rangle < \langle Q, N_k \rangle$$

While this tells us that which node is Q more similar to, it does not give us a numeric value for similarity, which is what we need if there is a taxonomic attribute in the case and the query.

One measure of similarity in a taxonomy could be derived by treating the length $d(N_i, N_k)$ of the shortest path between two nodes as a measure of distance between the two nodes. Then, similarity between the two nodes N_i and N_k is,

$$\text{sim}(N_i, N_k) = 1 - d(N_i, N_k)/D$$

where D is the length of the maximum path in the taxonomy. However, this similarity measure has a problem that it is not sensitive to where the two nodes lie in the taxonomy. It assigns the same similarity to the pairs (Khao-Pad, Red-Curry), (Asian, European), and (Dinner, British). A related measure known as *lch* defines it as follows (Leacock and Chodorow, 1998),

$$\text{sim}_{lch}(N_i, N_k) = -\log(d(N_i, N_k)/2D)$$

An alternative definition of similarity takes into account where the two nodes lie. In particular, if there are two siblings deep in the taxonomy, they are more similar as compared to two siblings higher up. Thus, the similarity of the pair (Khao-Pad, Red-Curry) should be more than the similarity of the pair (Asian, European) even though both are pairs of siblings. One method called *wup* due to (Wu and Palmer, 1994) does this and computes the similarity between N_i and N_k as follows,

$$\text{sim}_{wup}(N_i, N_k) = 2d(\langle N_i, N_k \rangle, \text{root}) / (d(N_i, \text{root}) + d(N_k, \text{root}))$$

where *root* is the root node in the taxonomy and $\langle N_i, N_k \rangle$ is the lowest common ancestor of N_i and N_k .

The seasoned gastronome amongst the readers would comment that the taxonomy above does not represent her notion of similarity of food items. For example, the French *crepe* is quite similar to the Indian *dosa*, also similar to the Ethiopian *injera*, whereas they are far apart in the taxonomy. That is, in fact, true because a closer observation reveals that the similarity defined on the taxonomy is based on the region the food belongs to, and not on the food itself. So one might say that the above taxonomy represents *familiarity*, based on region, and foods from similar regions are considered to be similar (from the point of choosing a menu that serves us well enough though).

It has also been pointed out that the notion of similarity also depends upon the task that is being addressed. For example, when your friend says that “Thai” food is okay, and if a restaurant offers “Red-Curry”, the similarity between the two should be 1. This can be termed *any-value* semantics (Bergmann, 2002). In the above example, the value in the case

("Red-Curry") is a descendant of the value in the Query ("Thai"). Any value semantics could also occur when the value in the Query (say "Ravioli") is a descendant of a value in a case (say "Italian"). This is interpreted as the situation when your friend wants to eat "Ravioli" and a restaurant offers *all* "Italian" food. One could refer to the former as any-value semantics in a recommender system scenario, while the latter is any-value scenario in a diagnostic scenario. As a more apt example of the latter, consider the situation when you have a pain in your ear then there is good match for consultation with an ENT (ear, nose and throat) specialist. Observe that in the any-value semantics, one is saying that the internal node represents a set of leaf nodes, and one of them will match perfectly with the leaf in question. Thus, your friend's desire to eat "Thai" could be interpreted as a desire to eat either "Red-Curry" or "Khao-Pad" and if the restaurant supplies the former then it is a good match. Likewise, if a doctor is a specialist in ear problems, *and* nose problems, *and* throat problems, she is a perfect match if you have an ear problem.

There can also be scenarios where the any-value semantics is not applicable and there is *uncertainty* associated with the internal nodes in the taxonomy. This can be termed as *some-value* semantics. In the recommender system situation, let us say that at some time in the past you have eaten some "Thai" food with your friend. Now you want it again, but don't remember quite what it was. If you go to a restaurant which serves only "Red-Curry", one cannot say how well it matches your "query". One could then adopt an optimistic approach, evaluating similarity as the maximum of similarity of "Red-Curry" with all the leaves that are descendants of "Thai", which will turn out to be 1. This means that you are optimistic that it was indeed "Red-Curry" you wanted. Or one could take a pessimistic approach, choosing the lowest similarity of a leaf descendant with "Red-Curry". This assumes that the "Thai" hyponym you actually wanted was in fact the one most dissimilar to "Red-Curry". One could also choose the average in proportion to the probability of their occurrence. In the diagnostic situation, there could be uncertainty if say, you were not sure whether it is your ear, or your eyes that is causing your headache and the doctor happens to be an ENT specialist. Then she may or may not⁸ be able to diagnose your problem, so you cannot say you have a perfect match.

15.1.7 Global Similarity

Let a case be defined by the set of attributes,

$$V_{\text{att}} = \{A_1, A_2, \dots, A_n\}$$

Consider two cases C_1 and C_2 ,

$$C_1 = (v_{11}, v_{12}, \dots, v_{1n})$$

$$C_2 = (v_{21}, v_{22}, \dots, v_{2n})$$

where v_{ik} is the value of the k^{th} attribute of C_i . We have already seen how the local similarity between the two values for each attribute can be computed. The task now is to define how the collection of local similarities between attributes of C_1 and C_2 determine the global similarity of the two cases. Let the local similarities between the two cases be represented by (l_1, l_2, \dots, l_n) . Then, we need to define an aggregation function to determine the global similarity $\text{sim}(C_1, C_2)^9$,

$$\begin{aligned}\text{sim}(C_1, C_2) &= \Phi(\text{sim}(v_{11}, v_{21}), \text{sim}(v_{12}, v_{22}), \dots, \text{sim}(v_{1n}, v_{2n})) \\ &= \Phi(l_1, l_2, \dots, l_n)\end{aligned}$$

The aggregation function Φ determines how the similarities at the attribute level combine together to determine the overall similarity. Various aggregation functions can be defined. Keep in mind that the local similarity values are in the range $[0, 1]$. We look at some possible definitions of Φ .

Average

The most intuitive aggregation is to take the average value of the local similarities. This gives us the function,

$$\begin{aligned}\Phi_A &\equiv \text{average}((\text{sim}(v_{11}, v_{21}), \text{sim}(v_{12}, v_{22}), \dots, \text{sim}(v_{1n}, v_{2n}))) \\ &\equiv (\text{sim}(v_{11}, v_{21}) + \text{sim}(v_{12}, v_{22}) + \dots + \text{sim}(v_{1n}, v_{2n})) / n\end{aligned}$$

Weighted Average The average aggregation function gives equal importance to all the attributes. In many problems, some attributes may be more important than others. A weighted average aggregation takes additional input in the form of a set of weights for the attributes that determines the *relative¹⁰importance* of each attribute. Let the set of weights be $W = (w_1, w_2, \dots, w_n)$. Then, the weighted average aggregation function is,

$$\Phi_{WA} \equiv (w_1 \cdot \text{sim}(v_{11}, v_{21}) + w_2 \cdot \text{sim}(v_{12}, v_{22}) + \dots + w_n \cdot \text{sim}(v_{1n}, v_{2n})) / \sum_1^n w_i$$

If the weights are normalized such that,

$$(w_1 + w_2 + \dots + w_n) = 1$$

then the weighted average aggregation becomes,

$$\Phi_{WA} \equiv (w_1 \cdot \text{sim}(v_{11}, v_{21}) + w_2 \cdot \text{sim}(v_{12}, v_{22}) + \dots + w_n \cdot \text{sim}(v_{1n}, v_{2n}))$$

The weights determine the relative importance of each attribute. For example, consider a domain in which the cases represent cylindrical objects using the attributes that include, amongst others, height, diameter and colour. Figure 15.12 shows two candidate objects that could match a query object. Only the RGB values for the colours are shown in the figure.

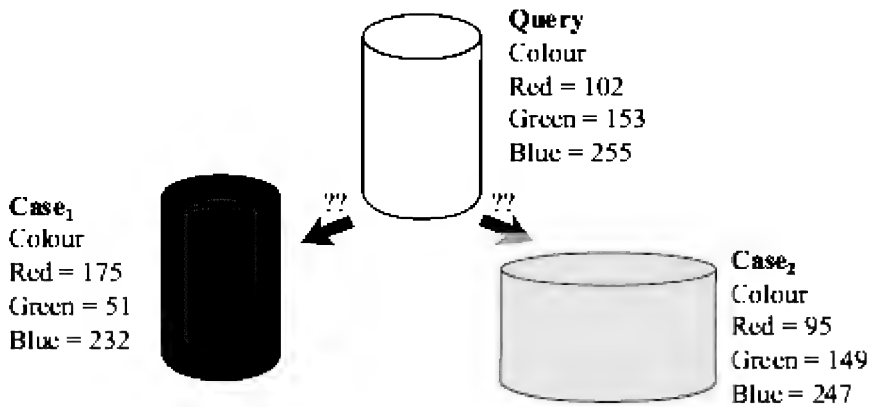


FIGURE 15.12 Which cylinder is more similar to the query? If colour has higher weight, then it is Case₂. If shape (height and diameter) has higher weight, then it is Case₁.

As one can see in the above diagram, Case₁ has a shape that is closer to the query than Case₂, while the latter has a colour that better matches the query colour. So which case is more similar to the query?

The answer depends upon whether shape is more important in the matching process, or colour. This relative importance can be represented by the weights in the case structure.

The weights could be assigned in different ways.

1. The weights could be static, and applicable to all the cases. Whenever a query is generated, these weights are used for computing similarity with all candidate cases.
2. The weights could be stated along with the query. Thus, each query also determines which attributes are more important. In a recommender system for example, the user could emphasize the features that she considers more important. For example, when looking for a flat to rent, one could give more importance to certain features like location, or number of balconies.
3. The weights could be case specific. This means that each case “knows” what attributes are more important when it is being considered. For example, if one were to build a case based medical diagnosis system then each candidate case (diagnosis) would state what features are more important for it to be true.

Maximum

Another simple aggregation function takes the maximum of the local similarities,

$$\Phi_{\max} \equiv \text{maximum}((\text{sim}(v_{11}, v_{21}), \text{sim}(v_{12}, v_{22}), \dots, \text{sim}(v_{1n}, v_{2n})))$$

or the weighted local similarities,

$$\Phi_{w\max} \equiv \text{maximum}(w_1 \cdot \text{sim}(v_{11}, v_{21}), w_2 \cdot \text{sim}(v_{12}, v_{22}), \dots, w_n \cdot \text{sim}(v_{1n}, v_{2n}))$$

The (weighted) maximum aggregation brings a case into contention, even if only one of its attributes matches very well. One can obtain progressively more demanding similarity functions by including the second highest local similarity (Max-2), the third highest (Max-3), and so on. Formally, let us arrange the local similarities between the two cases in decreasing order of magnitude, (h_1, h_2, \dots, h_n) . We have renamed the values to h_i only to highlight the fact that the list contains the highest, the second highest and so on in decreasing order. The different aggregation functions can then be defined as,

$$\Phi_{\max-k} \equiv h_k$$

Thus, *Max-k* aggregation says that one should look at the similarity of the k^{th} attribute when they are arranged in decreasing order of local similarity. Since the cases are selected based on their global similarity score, another way of looking at *Max-k* aggregation is that at least k attributes should have a high match.

Minimum

The minimum aggregation function is the opposite of maximum aggregation.

$$\Phi_{\min} \equiv \text{minimum}((\text{sim}(v_{11}, v_{21}), \text{sim}(v_{12}, v_{22}), \dots, \text{sim}(v_{1n}, v_{2n})))$$

or the weighted local similarities,

$$\Phi_{\min} \equiv \text{minimum}(w_1 \cdot \text{sim}(v_{11}, v_{21}), w_2 \cdot \text{sim}(v_{12}, v_{22}), \dots, w_n \cdot \text{sim}(v_{1n}, v_{2n}))$$

The (weighted) minimum aggregation is the strictest possible aggregation function. It says that the global similarity is equal to the lowest local similarity. Even if one attribute does not match well, the case goes out of contention. All attributes should match well with this aggregation function. Analogous to the *Max-k* aggregation, we can also define *Min-k* aggregation functions that are progressively less strict. The formal definition is left as an exercise for the reader.

The global similarity depends upon the local similarities of the attributes and also the weights assigned to the attributes. Let us consider a small fictitious case base of employees in a fictitious company called 'Anokhi Research'. The case base is given in Table 15.2 below. Let us assume the task is to predict the salary of a new (query) candidate, based on the salaries of the most similar candidates.

The case schema contains the following attributes,

- Name: Employee name. Not used in the similarity function.
Weight = 0
- Gender: "M" or "F". Uses equality as similarity.
Weight = 10
- Age: Numeric. Uses a linear similarity function with range 50.
Weight = 10
- Experience: Numeric. Linear function with range 50.

Weight = 10

- Education: Unordered symbol. Similarity table.
Weight = 20
- HandOn?: Boolean. Uses equality as similarity.
Weight = 10
- Salary: Solution attribute. Numeric.

The attribute “Education” takes three values: “Bachelors”, “Masters” and “PhD”. One could have defined this as an ordered symbol type. Choosing a user defined table however, allows us to define the local similarities explicitly. We define the following similarity table for the attribute Education.

Table 15.2 A sample similarity table for unordered type attributes

<i>sim_{Education}</i>	<i>Bachelors</i>	<i>Masters</i>	<i>PhD</i>
Bachelors	1	0.75	0.5
Masters	0.75	1	0.75
PhD	0.50	0.75	1

Table 15.3 A small fictitious case base

<i>Name</i>	<i>Gender</i>	<i>Age</i>	<i>Experience</i>	<i>Education</i>	<i>HandsOn?</i>	<i>Salary</i>
Abayomi	M	31	3	PhD	No	3400
Abdul	M	26	2	Masters	Yes	6500
Abheek	M	32	8	Masters	No	7400
Abigail	F	34	11	Bachelors	No	3650
Abner	M	46	24	Bachelors	No	14600
Acastus	M	46	17	PhD	No	13950
Adorna	F	29	4	Masters	No	2800
Adria	F	31	4	PhD	No	3700
Adrian	M	36	6	PhD	Yes	17600
Agatha	F	47	23	Masters	No	14650
Agnar	M	38	11	PhD	Yes	18600
Agudi	F	43	20	Bachelors	No	14000
Ahneta	F	42	18	Bachelors	Yes	19600
Aimara	F	50	23	PhD	Yes	21000
Airyaman	M	34	12	Bachelors	Yes	12800
Aithne	F	54	30	Masters	Yes	22200
Akulina	F	24	2	Bachelors	Yes	2300
Akira	M	51	26	Masters	Yes	21400

Aleron	M	34	6	PhD	Yes	18000
Alex	M	34	9	Masters	Yes	12550
Alice	F	28	5	Masters	No	2950
Alyssa	F	35	11	Masters	No	7850
Amalie	F	25	3	Bachelors	No	2450
Amika	F	29	7	Bachelors	Yes	12050
Amirthini	F	46	16	PhD	Yes	20400
Anana	F	30	1	PhD	Yes	16200
Anantamati	M	55	31	Masters	No	15450
Anbuselvan	M	37	9	PhD	No	8900
Andreas	M	24	1	Bachelors	Yes	2150
Angela	F	22	1	Bachelors	No	2150
Anisah	F	39	10	PhD	No	10200
Anta-Anclla	F	35	7	PhD	Yes	18500
Anton	M	33	8	Masters	No	7400
Anurag	M	25	2	Masters	No	2600
Anuragini	F	25	0	Masters	Yes	6200
Anunni	M	35	12	Bachelors	No	3800
Anzhela	F	37	14	Masters	Yes	13300
Archana	F	50	22	PhD	No	14700
Arezoo	F	30	8	Bachelors	No	3200
Aryenish	F	28	3	Masters	Yes	6650
Ashraf	M	32	3	PhD	Yes	17000
Ashutosh	M	46	21	Masters	Yes	20400
Atahualpa	M	31	9	Bachelors	No	3350
Atsu	M	23	0	Bachelors	No	2100
Aurang	M	37	14	Bachelors	Yes	13100
Ayodele	M	40	17	Bachelors	Yes	19400
Azibo	M	32	3	PhD	No	3650
Azuma	F	26	4	Bachelors	Yes	2600

Let us look at a few queries and the three best matching cases.

Query Example 1

The table below represents the query. The problem is to predict the salary that Ayesha is likely to get.

<i>Name</i>	<i>Gender</i>	<i>Age</i>	<i>Experience</i>	<i>Education</i>	<i>HandsOn?</i>	<i>Salary</i>
Ayesha	F	25	4	Masters	Yes	?

The best three matching cases in our case base ordered on similarity are

	<i>Name</i>	<i>Gender</i>	<i>Age</i>	<i>Experience</i>	<i>Education</i>	<i>HandsOn?</i>	<i>Salary</i>
1	Anuragini	F	25	0	Masters	Yes	6200
2	Aryenish	F	28	3	Masters	Yes	6650
3	Anzhela	F	37	14	Masters	Yes	13300

The similarity values are given below with the last column being the global similarity:

1	0	1	1	0.92	1	1	0.99
2	0	1	0.94	0.98	1	1	0.99
3	0	1	0.76	0.8	1	1	0.93

One way to compute the expected or predicted salary of the new candidate is to take the weighted average of the retrieved salaries. The weights are the similarity values. Based on this, the expected salary of Ayesha is,

$$\text{Salary}_{\text{expected}} = (6200 \times 0.99 + 6650 \times 0.99 + 13300 \times 0.93) / (0.99 + 0.99 + 0.93) \\ = 8622$$

The reader would have observed that the retrieved cases match exactly on Gender, Education and HandsOn? attributes, and for those attributes the local similarity is 1.

Let us try a new weight schema which gives less weight to Gender and more to Age, as shown below.

weight	1	25	10	20	10
--------	---	----	----	----	----

The new retrieval set is,

1	Anuragini	F	25	0	Masters	Yes	6200	Medium
2	Aryenish	F	28	3	Masters	Yes	6650	Medium
3	Abdul	M	26	2	Masters	Yes	6500	Medium

with similarities,

1	0	1	1	0.92	1	1	0.99
2	0	1	0.94	0.98	1	1	0.97
3	0	0	0.98	0.96	1	1	0.97

Observe, that while Anuragini and Aryenish still occupy the first two positions, Anzhela has been replaced by Abdul. Also note that the predicted salaries are more similar in the three retrieved cases, yielding,

$$\text{Salary}_{\text{expected}} = (6200 \times 0.99 + 6650 \times 0.97 + 6500 \times 0.97) / (0.99 + 0.97 + 0.97) \\ = 6449$$

What has happened in the second set is that because we have reduced the weight of *Gender*, and increased the weight of *Age*, a different set of cases has a higher similarity value. The three cases are similar in all other respects and also predict a similar salary. This suggests that in our (fictitious) case base, gender does not matter. This fact will be borne out when we construct a decision tree for the given case base in a later chapter. Let us try another query.

Query Example 2

The second query is

<i>Name</i>	<i>Gender</i>	<i>Age</i>	<i>Experience</i>	<i>Education</i>	<i>HandsOn?</i>	<i>Salary</i>
Azizi	M	43	15	PhD	No	?

The retrieval set with the *second* set of weights is,

	<i>Name</i>	<i>Gender</i>	<i>Age</i>	<i>Experience</i>	<i>Education</i>	<i>HandsOn?</i>	<i>Salary</i>
1	Acastus	M	46	17	PhD	No	13950
2	Anisah	F	39	10	PhD	No	10200
3	Anbuselvan	M	37	9	PhD	No	8900

with similarities,

<i>Name</i>	<i>Gender</i>	<i>Age</i>	<i>Experience</i>	<i>Education</i>	<i>HandsOn?</i>	<i>Similarity</i>
0	1	0.94	0.96	1	1	0.97
0	0	0.92	0.9	1	1	0.94
0	1	0.88	0.88	1	1	0.94

And the predicted salary is 11052.

Observe again that a different gender employee has come into the three most similar cases. We can verify that this is because of the low weight for *Gender*. Let us look at the retrieval set with the first set of weights we began with.

The retrieval set with original weights is,

	<i>Name</i>	<i>Gender</i>	<i>Age</i>	<i>Experience</i>	<i>Education</i>	<i>HandsOn?</i>	<i>Salary</i>
1	Acastus	M	46	17	PhD	No	13950
2	Anbuselvan	M	37	9	PhD	No	8900
3	Azibo	M	32	3	PhD	No	3650

with similarity values,

	<i>Name</i>	<i>Gender</i>	<i>Age</i>	<i>Experience</i>	<i>Education</i>	<i>HandsOn?</i>	<i>Similarity</i>
1	0	1	0.94	0.96	1	1	0.98
2	0	1	0.88	0.88	1	1	0.96
3	0	1	0.78	0.76	1	1	0.92

The predicted salary is 8941.

True enough. When *Gender* is given a significant weight, the same gender cases get a higher similarity value. One can also observe that the predictions in the best three cases are wildly different, even though the similarity values themselves are quite high.

Thus, high similarity by itself does not mean anything. Because whatever the similarity function we define, some cases will turn up with high values. What *is* important is to define the similarity function such that the solution of the retrieved case has *high utility*. In our example, this means that the predicted salary is as accurate as possible. If that can be done then CBR will work. Similar candidates will have similar salary predictions. This can be achieved in our problem by defining appropriate local similarities and also appropriate weights.

The case structure represented here may be termed *flat*, because it is a

collection of attributes without any other constraints. Flat cases capture the essence of structural CBR, viz. that cases are made of components that have their own local similarities, and these local similarities can be aggregated to compute global similarity.

Flat cases are easy to implement. One can store them in relational database systems, and this allows us to build commercial systems that contain a large number of cases.

However, subsets of attributes may be clumped together and can have their own identity. Different clumps in a case may have some relation with each other, and the case can have its own internal structure. This could have further implications on similarity computation. We shall look at such an object oriented representation later in the chapter. Below, we define a measure that may give a CBR system designer a means to evaluate similarity functions.

15.1.8 Case Cohesion

A similarity measure is good if the cases with high similarity also have high utility. In some domains, one can also define a similarity measure for cases using the lesson component. Then, one could define an evaluation measure called *cohesion*, originally defined for textual, case based reasoning (Lamontagne, 2006). The basic idea is to look for overlap between the set of most similar cases using the description side similarity, with the cases on the solution side similarity. The overlap between the two sets would determine the cohesion of the case base. Figure 15.13 below illustrates the concept.

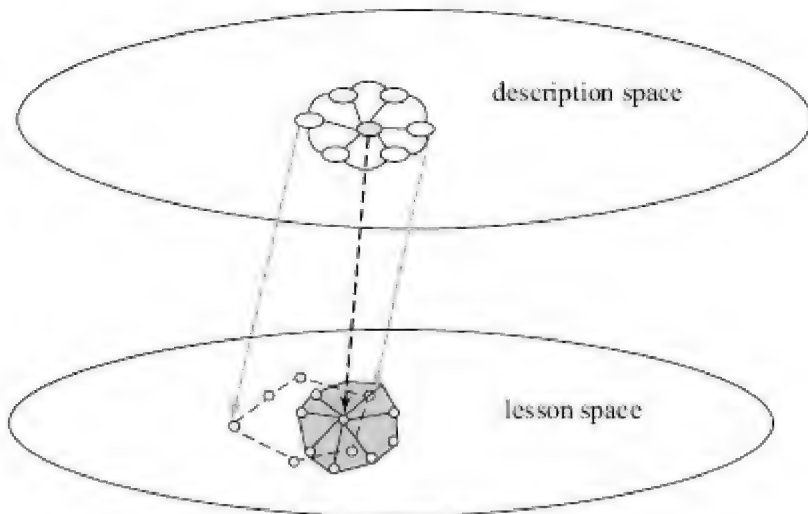


FIGURE 15.13 Cohesion is defined by the overlap between the set of cases that are the most similar to a case on the problem side similarity, and the set of cases that are most similar on the solution side similarity.

Let the two sets S_D and S_L be constructed using thresholds t_D and t_L using similarity functions sim_D and sim_L . That is,

$$S_D(C, CB) = \{\text{case} \in CB \mid sim_D(\text{case}, C) > t_D\} \text{ and}$$

$$S_L(C, CB) = \{\text{case} \in CB \mid sim_L(\text{case}, C) > t_L\}$$

Case cohesion can then be defined as,

$$cohesion(C) = S_{inter}(C, CB) / S_{union}(C, CB)$$

$$S_{inter}(C, CB) = S_D(C, CB) \cap S_L(C, CB) \text{ and}$$

$$S_{union}(C, CB) = S_D(C, CB) \cup S_L(C, CB)$$

If the two sets are identical then the value of case cohesion will be 1. A lower value means that there are more cases that belong to either $S_D(C, CB)$ or $S_L(C, CB)$, but not to the other.

One can compute the cohesion of the case base by computing the case cohesion value for each case. The higher the value of cohesion means that the best matching cases on the problem space similarity are likely to be the best matching cases on the solution space similarity. That is, similar problems will have similar solutions.

Now we turn our attention to *how* the best matching cases may be retrieved from a case base.

15.2 Retrieval

The task of retrieval is to retrieve the best matching cases. As discussed earlier, in this chapter, this may involve,

- retrieving all cases with maximum similarity.
- retrieving K Nearest Neighbours (KNN) or the K most similar cases.
- retrieving all cases above a given threshold level of similarity.

15.2.1 Similarity Based Retrieval

Similarity search, or nearest neighbour search, is different from range search in databases. Suppose we have some four dimensional numeric data in the range [0 –100]. Then, given a query with an attribute schema (A_1, A_2, A_3, A_4) ,

$$Q = (30, 40, 50, 60)$$

the following cases have equal similarity using the average aggregation as shown below:

$$\begin{aligned}
C_1 &= (40, 20, 40, 50) \\
C_2 &= (50, 80, 50, 60) \\
C_3 &= (80, 40, 50, 60) \\
sim(Q, C_1) &= \text{average}((sim(q_1, c_{11}), sim(q_2, c_{12}), sim(q_3, c_{13}), sim(q_4, c_{14}))) \\
&= (sim(q_1, c_{11}) + sim(q_2, c_{12}) + sim(q_3, c_{13}) + sim(q_4, c_{14}))/4 \\
&= ((1 - |30 - 40|/100) + (1 - (|40 - 20|/100)) + (1 - (|50 - 40|/100)) + (1 - (|60 - 40|/100)))/4 \\
&= (0.9 + 0.8 + 0.9 + 0.8) / 4 \\
&= 0.85 \\
sim(Q, C_2) &= (sim(q_1, c_{21}) + sim(q_2, c_{22}) + sim(q_3, c_{23}) + sim(q_4, c_{24}))/4 \\
&= ((1 - |30 - 50|/100) + (1 - (|40 - 80|/100)) + (1 - (|50 - 50|/100)) + (1 - (|60 - 60|/100)))/4 \\
&= (0.8 + 0.6 + 1.0 + 1.0) / 4 \\
&= 0.85 \\
sim(Q, C_3) &= (sim(q_1, c_{31}) + sim(q_2, c_{32}) + sim(q_3, c_{33}) + sim(q_4, c_{34}))/4 \\
&= ((1 - |30 - 90|/100) + (1 - (|40 - 40|/100)) + (1 - (|50 - 50|/100)) + (1 - (|60 - 60|/100)))/4 \\
&= (0.4 + 1.0 + 1.0 + 1.0) / 4 \\
&= 0.85
\end{aligned}$$

While C_1 differs from the query Q in all four attributes, C_2 differs only on two, and C_3 only on one attribute. But the one attribute C_3 differs on, is by a large amount. It would thus be difficult to keep a range on the values of the attributes for retrieval. Suppose we kept a range (+/- 10, +/- 10, +/- 10, +/- 10) then only C_1 would be retrieved, even though the other two are considered equally similar. On the other hand, if we kept a larger range say, (+/- 40, +/- 40, +/- 40, +/- 40), then some cases with low similarity will also be retrieved. For example,

$$C_4 = (70, 80, 10, 20)$$

would be retrieved, even though it has a similarity value of 0.6.

Thus, while similarity search does allow the deviation of attribute values from the query values, it does not specify the amount of deviation for individual attributes. If one were to be doing threshold-based retrieval, the constraints are on the aggregated similarity value rather than on individual attribute similarities.

We begin by describing how the KNN retrieval can be done sequentially.

15.2.2 Sequential Retrieval

The simplest algorithm is to look at all the cases sequentially and maintain a retrieval set R that satisfies the required retrieval criteria. Figure 15.14 below describes the algorithm at a high level. The retrieval set R is maintained as a list of pairs, where each pair contains the case id and the value of similarity with the query Q . Let the case base contain N cases, let K and N be input parameters. The first line of the algorithm does a cursory check on the size N . If it is less than K then the entire case base is returned, sorted on similarity with the query Q . Otherwise, the first K cases are inserted into R . The rest of the cases are compared with Q sequentially, and if found to be better, replace the lowest similarity case in R .

```

SequentialKNN(q : query, CB : case base, k : retrieval size, n : case base
size)
1  if k ≥ n
2    then return Sortsim(CB)
3  R ← {}
4  for i ← 1 to k
5    do Insert (< casei, Sim(casei, q) >, R)
6  kSim ← Second(First(R))
7  for i ← (k + 1) to n
8    do s ← Sim(casei, q)
9      if s > kSim
10        then R ← Insert (< casei, s>, Rest(R))
11        kSim ← Second(First(R))
12 return Reverse(R)

Insert(newPair, list)
1  if Second(newPair) ≤ Second(First(list))
2    then return Cons(newPair, list)
3  else return Cons(First(list), Insert(newPair, Rest(list)))

```

FIGURE 15.14 The Sequential KNN retrieval algorithm maintains a sorted list of K cases. When it finds a better case, it removes the first case with lowest similarity and inserts the new one in its correct place.

In the above algorithm, we have used the list functions “First” to return the first element of a list (called “car” in Lisp), “Second” to return the second element (defined as (car(cdr list)) in Lisp), “Rest” to return the tail of a list (called “cdr” in Lisp), and the Lisp function “Cons” that adds a new element to the head of a list. The function “Insert” adds a new pair to the list in its sorted place.

The retrieval set contains a set of K cases that have the highest similarity with the query Q . The algorithm given here is for the second criteria, or the KNN retrieval. This is the most commonly used criteria. The reader is encouraged to modify the algorithm for the other two criteria.

The complexity of the algorithm is linear in the number of cases, because each case is inspected once. It is also linear in the size of the retrieval set K , because insertion may involve comparison with K cases in R . Finally, it is dependent on the complexity of similarity computation. The similarity of each case with the query is computed once.

Sequential retrieval works fine in many simple applications, especially with increasing computing power. If one has a few thousand cases and retrieval takes a few seconds, it may be good enough (see for example (Khemani et al, 2002)). However, there may be problems where sequential retrieval is not fast enough. This may be because the number of cases is very large, and/or the similarity computation is very expensive.

We first look at some approaches to handle a large number of cases. We take up the simpler scenario where all attributes take values in a metric space. After that, we will look at case representations where similarity computation is expensive.

In both situations, there is a need to cut down on the number of similarity computations.

15.2.3 Metric Spaces and Nearest Neighbours

Cases made up entirely of numeric attributes can be handled by special methods because cases in metric spaces can be thought of as points in an N -dimensional space, where N is the number of attributes. One can then define similarity as a function of a global distance between the two cases. Given two numeric attributed cases C_1 and C_2 ,

$$C_1 = (n_{11}, n_{12}, \dots, n_{1N})$$

$$C_2 = (n_{21}, n_{22}, \dots, n_{2N})$$

the following distance measures can be defined between the two cases.

Manhattan Distance

The Manhattan or City Block distance is a sum of the distance along each dimension. It is so named because that is the amount one would have to walk on a square grid of roads in many modern cities.

$$dist_{Manhattan}(C_1, C_2) = \sum_{i=1, N} |n_{1i} - n_{2i}|$$

One can also compute the weighted Manhattan distance as,

$$dist_{WManhattan}(C_1, C_2) = \sum_{i=1, N} w_i * |n_{1i} - n_{2i}|$$

Weights could be incorporated into the distance measures described below as well. A commonly used distance measure in metric spaces is the Euclidean distance measure.

Euclidean Distance

$$dist_{Euclidean}(C_1, C_2) = \sum_{i=1, N} (n_{1i} - n_{2i})^2)^{1/2}$$

The Euclidean distance measures the length of the straight line between the two points, representing the cases in N -dimensional space.

The above two measures in some sense aggregate the differences at the attribute level. We can also replicate the Max and Min similarity by corresponding distance measures.

Min Distance

$$Dist_{min}(C_1, C_2) = \min(|n_{1i} - n_{2i}|)$$

The Min distance measure is the smallest difference in any dimension between the two cases. Observe that it would correspond to the similarity aggregation function Φ_{max} .

Max Distance

$$Dist_{max}(C_1, C_2) = \max(|n_{1i} - n_{2i}|)$$

The Max distance measure treats the largest difference as the distance

between the two cases.

The following distance measure is a generalization of many of the functions described above.

Minkowski Norm

$$Dist_{Minkowski}(C_1, C_2) = (\sum_{i=1}^N |n_{1i} - n_{2i}|^P)^{1/P}$$

The choice of the parameter P determines the nature of this distance measure. When $P = 1$, it becomes the Manhattan distance; with $P = 2$ it is the Euclidean distance and as $P \rightarrow \infty$ it becomes the Max distance function.

Seen in this N -dimensional space, the closer the two cases are the more similar they are. If our retrieval task is to find the most similar cases, then we can use the distance measure directly. The name *nearest neighbour* in fact reflects this criterion.

If we need a similarity value, for example in threshold based retrieval then we need to map the distance value to a similarity value. While doing so, we need to know the maximum possible distance D_{MAX} that two cases can have in the N -dimensional space. Then,

$$sim(C_1, C_2) = 1 - dist(C_1, C_2)/D_{MAX}$$

Another way would be to normalize the attribute level distances to the range $[0,1]$ and define the global distance function appropriately. This is left as an exercise for the reader.

Ordered symbol type attributes where the values are assumed to be spaced equally apart can also be treated as having a distance value. We can even extend the idea to Boolean attributes, with the distance between two values being limited to the set $\{0,1\}$.

15.2.4 KD Trees

When the case base becomes very large, there is a need to apply some techniques to selectively inspect a subset of promising cases. The reader must be familiar with binary search trees and B-trees (Cormen et al., 2001). We look at a tree called *kd-tree* that was first devised by Bentley (1975). The *kd-tree* is different from binary search trees in two ways. One is that binary search trees search on one-dimensional data, while *kd-trees* search over multi-dimensional data. Two, *kd-trees* are designed for similarity searching, as opposed to the equality based searching in binary search trees. However, both operate on numeric spaces.

When cases can be seen as points in N -dimensional space, one can imagine the query as another point in the same space. The task then is to retrieve the cases that are closest to the query. If one can partition this space into regions that can be accessed by hierarchical index structures, then one could satisfy the retrieval criteria by inspecting only a few regions. Each region would contain cases similar to each other, and hence roughly

equally similar to the query. This is the basic idea behind many retrieval algorithms in metric spaces (see for example (Frakes and Baeza-Yates, 1992), (Nene and Nayar, 1996), (Kleinberg, 1997), (Kushilevitz et al., 1998), (Van Berendonck and Jacobs, 2003)).

A *kd-tree* is a multidimensional binary search tree. Each internal node in the *kd-tree* has a test for the value of some attribute, like in a binary search tree. Each internal node represents a set of cases, and the two arcs below it point to two partitions of that set discriminated on the answer to the test. There is no restriction on the number of times an attribute can be used as a test. That is dictated by the distribution of cases in the *N*-dimensional space. Each leaf node represents a bucket that stores a set of cases. The size of the bucket is user determined. All the cases inside the bucket are inspected sequentially and that is a factor in determining the bucket size. Figure 15.15 shows a sample *kd-tree* constructed on a synthetic case space on two attributes X_1 and X_2 . These two attributes may be the only ones defining the case, or may be the only (numeric) ones selected for constructing the tree.

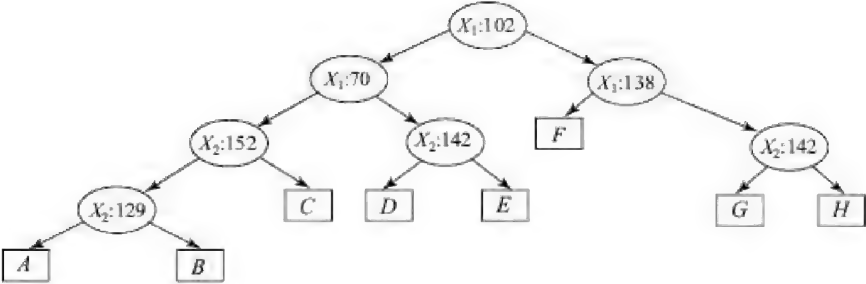


FIGURE 15.15 A sample *kd-tree* for the data shown in Figure 15.16. The data is partitioned into eight buckets named A to H. The number in the node represents the test (value & number). Cases answering “yes” take the left branch.

The *kd-tree* above has been constructed to partition the cases in the two-dimensional space shown in Figure 15.16. Each dot represents a case in the figure, with its coordinates being the values of the two attributes.

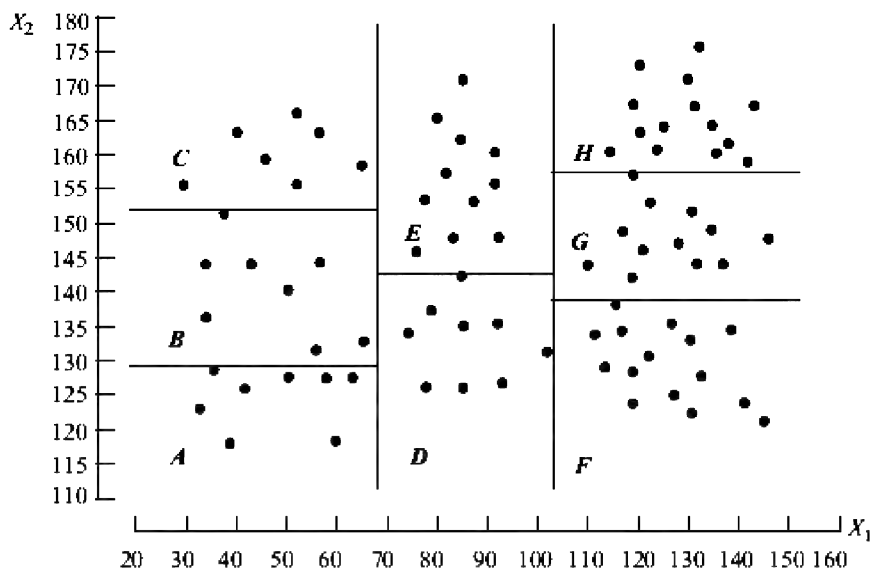


FIGURE 15.16 The case base in two-dimensional space defined by the attributes X_1 and X_2 . The first partition is marked by numeral 1, and is based on a value 102 for attribute X_1 . The labels A–H correspond to the leaves in the *kd-tree* in Figure 15.15.

The *kd-tree* is built by a recursive algorithm in a top down manner. It begins with a single node at the root representing the entire case base. It then partitions the set of cases represented by the node, based on some attribute, and recursively calls itself to build the subtrees. The process halts on some termination criteria, which could be the size of the bucket or a threshold average intra-bucket similarity. The algorithm for building the *kd-tree* is described in Figure 15.17 below. The structure of the nodes is,

Node \leftrightarrow (attribute-name,	: the attribute to be tested
attribute-value,	: the value of the attribute
pointer-to-bucket,	: bucket has the cases
pointer-to-left-child,	: cases with smaller attribute values
pointer-to-right-child,	: cases with larger attribute values
pointer-to-parent,	: pointer to parent node
count).	: number of times visited during search

```

Build-kdtree(CB : case base, A : attribute list, b : bucket size, parent :
parent node)
1  if Size(CB) ≤ b
2  then  lSubtree ← NIL
3         rSubtree ← NIL
4         b ← MakeBucket(CB)
5         return p ← (NIL, NIL, b, lSubtree, rSubtree, parent, 0)
6  else
7         testAttr ← ChooseAttr(A, CB)
8         testVal ← ChooseVal(testAttr, CB)
9         largerCB ← LargerThan(CB, testVal)
10        smallerCB ← CB
11        b ← NIL /* not a bucket */
12        p ← emptyNode
13  return p ← (testAttr, testVal, b,
14              Build-kdtree(smallerCB, A, b, p),
15              Build-kdtree(largerCB, A, b, p), pNode, C)

```

FIGURE 15.17 The algorithm Build-kdtree returns a pointer P to a node of the type (attribute-name, attribute-value, pointer-to-bucket, pointer-to-left-child, pointer-to-right-child, pointer-to-parent count). The last element “count” is used in the retrieval phase. Function Size counts the number of cases in CB. Function *ChooseAttr* selects and returns the attribute to be used for test at that node. Function *ChooseVal* selects and returns the test value for the chosen attribute. The function *LargerThan* selects the subset of CB that has cases with the value of the chosen attribute larger than the chosen value T . Function *MakeBucket* creates a bucket of cases and returns a pointer to the bucket.

The function is called with the value of the parent argument as “nil” to mark the root node. We create the empty node ($P \leftarrow \text{emptyNode}$) so that children in the recursive call can establish a pointer to the parent node.

The termination criterion in the above algorithm is the bucket size. This dictates the time needed to search the bucket. The reader is encouraged to try out different criteria, like the average intra bucket similarity.

The first choice one has to make in the algorithm is to choose the attribute to test at a given node, and the value of that attribute that is used in the test. Let us assume that the test value for each attribute is the median, which divides the cases equally into two halves. The objective is to create two partitions, such that each contains cases that are more similar to each other. The following criteria are possible.

One method to do this is to compute the average global similarity between cases in the two partitions produced by each attribute. The attribute that yields the highest average similarity is used as a test at that node. Observe that this will be done at the time of building the *kd-tree* offline, and one presumes that one can afford more time. One can also construct a table of similarities before building the *kd-tree*.

Another method is to use the inter-quartile distance on each attribute as shown in Figure 15.18 below. The distance is computed only along one dimension by computing the difference only on the values of the concerned attribute. Let the case base be partitioned into four equal size quarters by separators along the dimension of a given attribute. The inter-quartile distance for an attribute is the distance between the first quarter and the fourth quarter. As shown in the figure, this distance is the maximum, along the dimension in which the spread of the cases is the maximum.

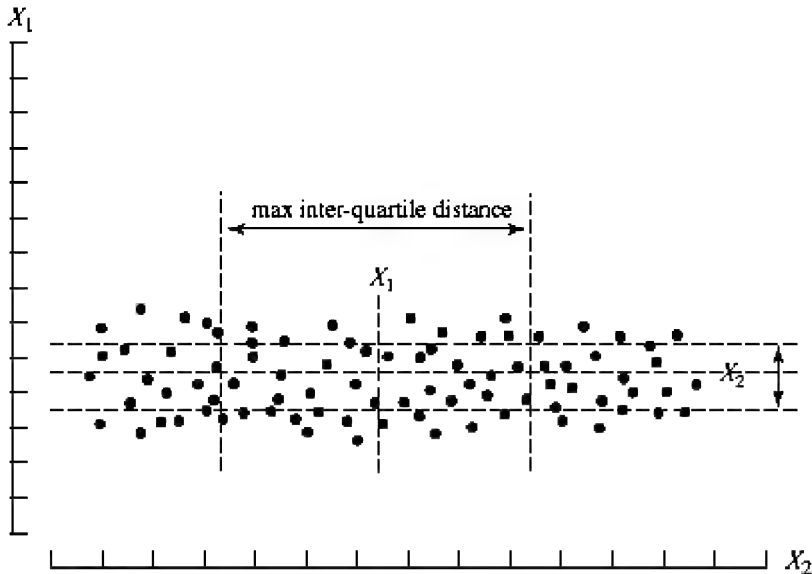


FIGURE 15.18 The vertical separator produced by the test on attribute X_1 produces partitions that are closely knit together, as compared to the test on X_2 . The maximum inter-quartile distance chooses to split across dimensions that have the *maximum spread*.

A third approach is to use the notion of *information gain* to choose the attribute. However, this is possible only when the cases are used to classify objects into different classes. We shall look at this approach when we study the *ID3* algorithm to build decision trees in Chapter 18.

The reader would have observed that constructing the *kd-tree* is an expensive process. Even for choosing one attribute, the entire set of cases has to be partitioned along all possible dimensions. However, one must keep in mind that this happens only in the preprocessing stage. The hard work put in during tree construction bears fruit later, during case retrieval, in the form of faster retrieval.

The first thing that happens when a query comes is that the retrieval algorithm goes down the *kd-tree* answering questions at each node. When it reaches a bucket, the cases in the bucket are compared sequentially with the query. Then, depending upon whether the retrieval criterion is satisfied or not, the algorithm may need to visit some neighbouring buckets. This is because a better case might lie in one of those buckets. That could happen for example, when the query falls near the corner of a bucket. The *kd-tree* retrieval algorithm employs two tests to make decisions on these issues.

The BWB Test

The *Ball Within Bounds* (BWB) test is used as the termination criteria.

The “ball” in question is a hypothetical surface defined by the retrieval criterion and the constructed retrieval set. For threshold based retrieval, it is defined by the threshold distance, or similarity. For KNN retrieval, it is

defined by the similarity of the K^{th} case in the retrieval set. The surface of the “ball” thus defined are points of equal distance (or similarity) from the query point. When we use Euclidean distance, as in the example below, the ball is a hypersphere in N -dimensional space. In two-dimensional space, as in the example, it is a circle. If one had used Manhattan distance, the circle would have been replaced by a rhombus. In N -dimensional space, the surface would be a hyper-rhombus.

The “bounds” are defined by the region in the space that has already been inspected. Thus, if the ball lies within the bounds of the region that has been inspected, it means that all cases in other regions are outside of the ball, and therefore will not meet the retrieval criteria. Remember, that cases outside the ball have lower similarity than the cases within the ball. If the BWB test returns *true*, the retrieval algorithm can terminate.

If the BWB test returns *false* then there is some part of the ball that lies in a region not yet explored. A better case could be lurking out there in that neighbouring region. The retrieval algorithm then backtracks up the tree one level and considers the other child. Should it go down that branch and visit the bucket(s) there? The answer is provided by the Ball Overlaps Bounds test.

The BOB Test

The *Ball Overlaps Bounds* test returns *true* if the ball overlaps the unexplored region in question. If the answer is *true*, then a more similar case might exist in the unexplored region. This is because the surface (boundary) of the ball marks the region of interest. For KNN retrieval, it marks the similarity, or distance, of the K^{th} case in the retrieval set. Any case at a lower distance, or higher similarity, will be of interest. When the BOB test returns *true*, the algorithm goes down the branch and (sequentially) compares cases in the bucket with the query, if it is a bucket. If it finds more similar cases then the ball will shrink in size as the better case enters the retrieval set.

The BOB test and the BWB test are illustrated in the figures below. The figures represent the same partition of the same case base in the example above, with the buckets represented by shaded rectangles. The rectangles are drawn as concentric figures to highlight the tree structure. In fact, the boundaries of a rectangle will coincide with the boundaries of its parent. Let us assume that the task is to retrieve 4 nearest neighbours for the query Q .

On processing a query Q , the retrieval algorithm ends up in bucket B . Figure 15.19 shows the query Q that falls in bucket B of the kd -tree. The four nearest cases are marked with white dots. The farthest of them defines the radius of the ball. The BWB ball test fails for the region inspected, which is bucket B .

The algorithm backtracks up the tree (see Figure 15.15) and considers B ’s sibling A . The BOB test succeeds for A , and the algorithm inspects the cases in bucket A . As shown in Figure 15.20 below, two new cases are found that go into the retrieval set. The ball shrinks in size, because the

farthest of the four is now closer than before. The BWB test now succeeds, and the algorithm can terminate. It has only inspected two of the eight buckets.

If the two cases found in bucket A had not existed then the ball would not have shrunk and the BWB test would have failed. The algorithm would have gone up and applied the BOB to the next bucket C. As one can see from Figure 15.19, this test would have failed. It would have gone further up and come down to bucket D after answering the test $X_2 \leq 142$. It would have applied the BOB test to bucket D with success, and proceeded to inspect the cases in that bucket. The point to note is that because of the BOB test, it would have skipped bucket C, and saved computation time.

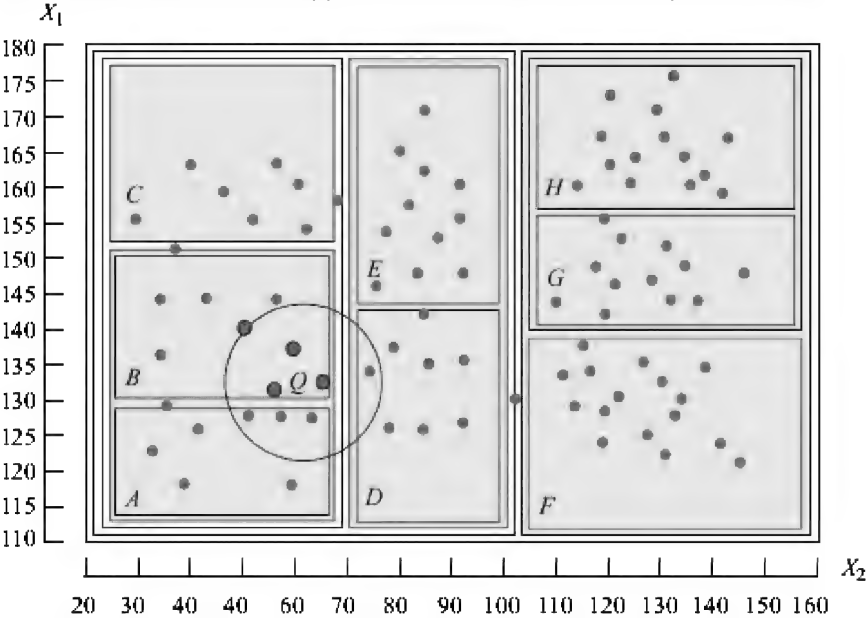


FIGURE 15.19 The query Q falls in the bucket B . The 4 nearest neighbours are shown as white dots. The distance to the fourth case marks the radius of the ball. As one can see, the BWB test fails. The algorithm goes up the tree and considers the other child A . The BOB test succeeds for A , and the algorithm explores bucket A .

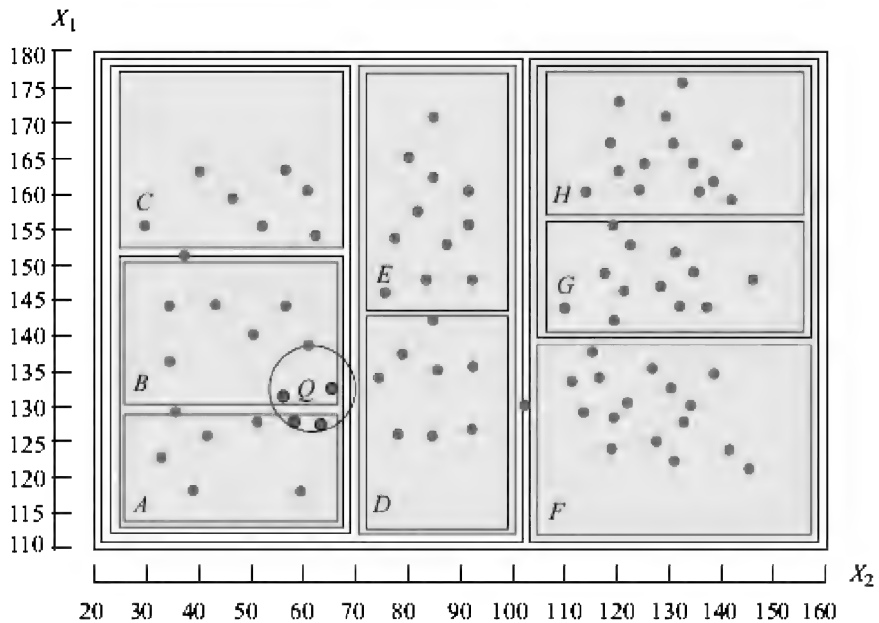


FIGURE 15.20 After the algorithm explores bucket A, it finds two more cases. The ball shrinks in size and the BWB test will succeed for the region defined by buckets A and B. The algorithm thus terminates with the cases in white.

How does one implement the BOB test and the BWB test? The BOB test needs to test whether the closest point in the region being considered is within the ball. Let us look at the two-dimensional space in Figure 15.21. There are essentially two kinds of points one needs to test the distance to. One when the closest point in the region, labelled 1 and 3 in the figure, is at the corner. The other is when the closest point is along one attribute dimension, as for region 2.

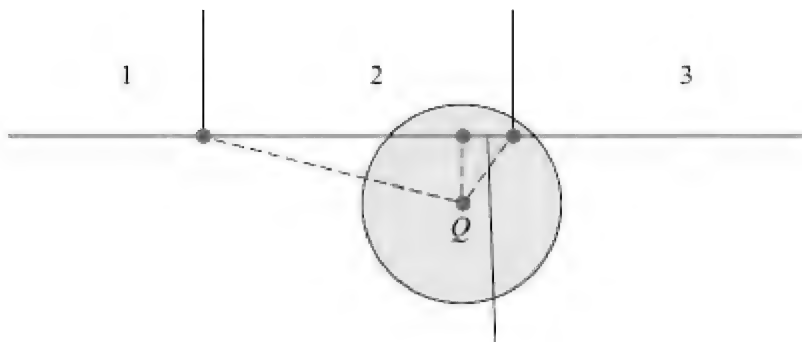


FIGURE 15.21 In a two-dimensional space, there are essentially two types of adjacent regions. The first is (regions 1 and 3), where the nearest point to the query is at a corner. The second of the type labelled 2 is where the nearest point is along a particular dimension. The BOB test needs to compare the lengths of the dotted segments as shown in the figure with the radius.

After identifying the point closest to the query in the region, the test

requires the computation of the length of the dotted segment and comparing it with the radius of the ball. With each new dimension added, one more type of point will be added. For example, in 3D space, the closest point could be a corner, on an edge, or on a face.

The BWB test is simpler. For each dimension, one needs to apply the test on both sides of the query point to check whether the distance to the separator is greater than the radius. The BWB test may be more complex for nonrectangular regions. If there are concave corners on the region boundary then the distance to those corners must also be compared with the radius as shown in Figure 15.22.

We assume that suitable BOB and BWB tests are implemented. It must be kept in mind that one is allowed to have error in these functions. The BWB test can be allowed to return *false* erroneously, and the BOB test can be allowed to return *true* erroneously. Functions that do that may be simpler to implement. They will not affect the correctness of retrieval. But the cost may be in terms of time taken for retrieval, due to unnecessary exploration of regions that a better function would have excluded.

The algorithm for retrieval is described in Figure 15.23.

The node structure in the *kd-tree* is repeated below.

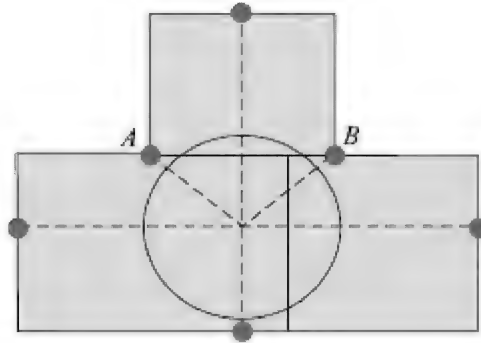


FIGURE 15.22 For nonrectangular shapes, the BWB test may be more complex. For example, if there are concave corners like A and B, they must also be compared.

Node	↔	(attribute-name,	:	the attribute to be tested
		attribute-value,	:	the value of the attribute
		pointer-to-bucket,	:	bucket has the cases
		pointer-to-left-child,	:	cases with smaller attribute values
		pointer-to-right-child,	:	cases with larger attribute values
		pointer-to-parent,	:	pointer to parent node
		count).	:	number of times visited during search

The function *UnvisitedChild*(*Q*, *A*, *node*) does the following,

1. It first looks at the “count” field in the node. If it has value 2 then the function returns the value “nil”. The value 2 signifies that both the children of the node have been visited.
2. Else it applies the test ($\text{value}_Q \notin \text{testValue}_{\text{testAttribute}}$) at the node to

the query Q .

3. If the answer is "true" then,
 - (a) If the value in the count field is 0, it is incremented to 1. Then the left child is returned as the child of the node.
 - (b) Else the value is incremented to 2, and the right child is returned as the child of the node.
4. If the answer is "false" then,

```

KNN-kdtree(q : query, root : kd-tree, A : attribute list, k : retrieval size)
1  R ← ( )
2  visited ← { }
3  node ← root
4  radius ← LARGE          /* a suitably large value */
5  while not BWB(radius, visited)
6    do while not Leaf(node)
7      do child ← UnvisitedChild(q, A, node)
8        if (child = NIL) OR (not BOB(q, node))
9          then node ← Parent(node)
10         if node = NIL
11           then return R
12         else node ← child
13       visited ← visited ∪ {node}          /* reached an eligible leaf */
14       CB ← CasesInBucket(node → B)      /* get cases */
15       for each c ∈ CB
16         do InsertK(<c, Sim(c, q)>, R, k)
17       radius ← Dist(q, First(R))
18       node ← Parent(node)
19       if node = NIL
20         then return R
21  return R

InsertK(pair, list, k)
1  if Size(list) < k
2    then /* insert anyway */
3      Insert(pair, list)
4    else /* insert only if better */
5      if Second(pair) > Second(First(list))
6        then Insert(pair, Rest(list)) /*discard head (weakest)
                                         element */

```

FIGURE 15.23 The KNN-kdtree algorithm retrieves the K most similar (nearest) cases from the leaves of the kd -tree. The function *UnvisitedChild* returns the child that best matches the Query, according to the test at that node, provided it has not been visited. In both, children have been visited it returns "NIL". The function *InsertK* inserts a case into a list if it is better than the lowest similarity value in the list, by calling the function *Insert* defined in Figure 15.14.

- (a) If the value in the count field is 0, it is incremented to 1. Then the right child is returned as the child of the node.
- (b) Else the value is incremented to 2, and the left child is returned as the child of the node.

Thus, the function either returns a child node, if it has not been visited, or it returns *NIL*.

Since the retrieval algorithm is modifying the kd -tree, it is necessary that for every query, a copy of the tree is made for searching.

The *BWB(Radius, Visited)* function should implement the Ball Within Bounds test on the region specified by the buckets visited. The function returns *false* when the set *Visited* is empty. In the version of the algorithm

above, the BOB test is applied whenever it tries to move to a child node. The BOB test should also return *true* when the node contains the query, so that the first time the search goes through to the leaf that contains the query. Thus, it should return *true* when the set *Visited* is empty. The function *Leaf(node)* tests whether the node is a leaf. This can be done by inspecting whether the pointer to the left (or right) child has value “NIL”. The function *Parent(node)* returns a pointer to the parent node. The pointer is available in the node. The function *CasesInBucket(nodeAEB)* returns the set of cases stored in the bucket *B* pointed to by the node. The function *InsertK(pair, list, K)* inserts a <node, *sim*(Q,node)> pair into the list. If the list has less than *K* pairs, it calls the function *Insert* defined in Figure 15.14. If the list has *K* pairs then it calls the function with the rest (or tail) of the list, removing the case with lowest similarity only if the new case is better. In either case, the new case is inserted (by function *Insert*) in its place in the sorted order. *Radius* is calculated as the distance between the query and the least similar case in the retrieval list *R*.

15.2.5 The Inreca Tree

The *kd-tree* is defined over numeric attributes. An extension of the *kd-tree*, known as the *Inreca* tree, was developed as part of the Inreca (INduction and REasoning from CAseS) projects (Althoff et al., 1998), (Bergmann et al., 1999), (Bergmann, 2001; 2002). The *Inreca* tree extends the *kd-tree* in two ways. One, it adds a branch for “unknown” below a node. The second is that it also caters to unordered symbol type attributes. Unlike the *kd-tree*, the *Inreca* tree is not a binary tree. The two types of nodes in the *Inreca* tree are illustrated below.

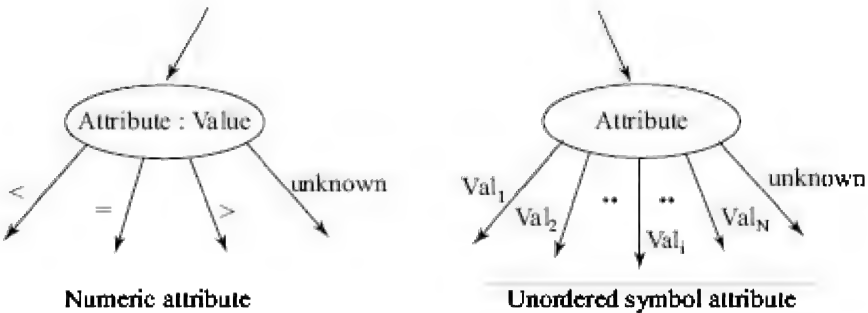


FIGURE 15.24 The two types of nodes in the *Inreca* tree. For numeric attributes, there may be an extra test for equality. For unordered symbol type attributes, there is an edge for every value. In addition, both types of nodes have an edge for unknown values as well. Figure adapted from (Althoff et al., 1998).

A question that arises is the notion of distance between cases when the attributes are not numeric. One will then have to define the distances between different values of an attribute. We have seen earlier that one can explicitly define similarity values for unordered symbol type attributes. We

can do so likewise for distance.

Once we think of cases as points in some space, we are essentially relating distance (inversely) to similarity. For that, one needs to find the maximum difference between values, to be equated to zero similarity value. As a corollary, a difference of say d in two different numeric attributes may have different impact. An extreme example is Boolean attributes. Here, a difference of 1 corresponds to (local) similarity of 0. In the example above of Figure 15.16, the X_2 axis has a maximum difference $70 = 180 - 110$, while the X_1 attribute has a maximum difference $140 = 160 - 20$. One way to address this is to normalize all distances to be in the range $[0,1]$. While doing so, one can also incorporate the weights of the different attributes.

The *kd-tree* and the *Inreca* tree employ a tree structure to index the cases in the case space. The objective is to group together cases similar to each other in compartments that are indexed. Other approaches that have employed indexing schemes are the structure GNAT (Brin, 1995), and the Bubbleworld (Efros, 1998).

15.2.6 Fish and Shrink Algorithm

The complexity of retrieval is dependent, both on the number of cases one has to search through, and the complexity of each similarity computation. The *kd-tree* and similar algorithms address the issue of inspecting a small number of cases from a large case base. The algorithm *Fish and Shrink*, on the other hand, addresses the issue when similarity function is very complex (Schaaf, 1995, 1996).

The similarity function can be complex when the case structure is complex. This can be when the cases are themselves structured. We give a brief idea here of when that can be. The interested reader is referred to (Bergmann, 2002) for a more detailed discussion. Consider, for example, a recommender system for computer systems. A case in this system is the description of a specific computer model. This can be an object oriented (OO) representation. In an object oriented representation, the relation between a device and its components is captured explicitly. For example, a computer system is made up of a processor, a monitor, a keyboard, the different kinds of primary storage like the RAM, the L1 and L2 cache, the secondary storage like the hard disk or flash drive, the auxiliary storage devices like DVD writers, and the different ports and network connectors. Each of these components are objects themselves, described by a collection of attributes like cost, make, speed and capacity. Also each of them may form a taxonomic hierarchy. For example, the auxiliary device may be further specialized into magnetic and laser devices, and the latter into the CD and the DVD. A user may want to compare two different machines. This would involve comparison of the collective set of attributes that define the case, as also the similarity that comes from the taxonomic organization of components. Given that there may be a large number of attributes whose local similarities have to be combined in complex ways,

and that there may be a large number of products to choose from, the task of computing similarity can be humungous. Another structure that can lead to very expensive similarity computations is when the case structure is a graph. This could be when one is reasoning with network traffic, transportation systems, or electricity and water supply networks.

The *Fish and Shrink* algorithm relies on the property that the similarity of similar cases to the query will be similar. As a corollary, if a case *A* is going to be out of contention for a place in a retrieval set then a case *B* that is close to *A* is likely to be out as well. The algorithm exploits this property as follows. Given a query *Q*, whenever the algorithm computes similarity with a test case *T*, it uses its knowledge of the similarity of *T* with other cases to update bounds on the similarity of all other cases with *Q*. Obviously, this will make sense only if the actual similarity computation is very expensive. If that is so, then the cost of visiting all cases and updating their similarity bounds will still be lower.

This property is better illustrated when we deal with distances, again, like we did during *kd-tree* construction. Figure 15.25 below shows the relation between the distances of a test case *T* from the query *Q*, to another case *C*. These bounds will hold *only* when the distance function obeys the triangle inequality.

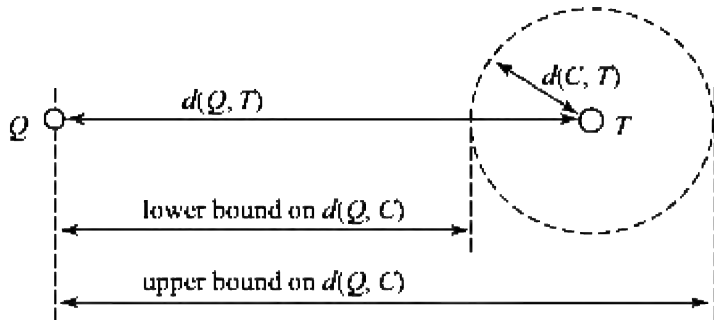


FIGURE 15.25 Triangle inequality. Given the pre-computed distance $d(C, T)$, the measurement $d(Q, T)$ bounds the distance between *Q* and *C*.

The triangle inequality says that given any three points *X*, *Y* and *Z*, the distances between the three satisfy the relations between the distances of the sides of a triangle. That is,

$$d(X, Y) + d(Y, Z) \geq d(X, Z)$$

The extreme case is when the three points are co-linear. Then,

$$d(X, Y) + d(Y, Z) = d(X, Z)$$

Using one extreme case (the “lunar eclipse” in the above figure) we get,

$$d(Q, T) + d(T, C) = d(Q, C)$$

the upper bound on $d(Q, C)$ as $d(Q, T) + d(T, C)$. The other extreme case

is when the point C is between Q and T , like in the solar eclipse,

$$d(Q, C) + d(T, C) = d(Q, T)$$

which gives us the lower bound on $d(Q, C)$ as $d(Q, T) - d(T, C)$ as shown in the figure. Thus,

$$d(Q, C) \geq d(Q, T) - d(T, C) \text{ and}$$

$$d(Q, C) \leq d(Q, T) + d(T, C)$$

A small point to note about the triangle inequality is that it holds when the distances are unbounded. If the distance were to be “normalized” to the range $[0,1]$ to define $d_{[0,1]}(X, Y)$, this relationship breaks down. For example, if the distance $d_{[0,1]}(Q, T)$ were to be 0.9, and $d_{[0,1]}(T, C)$ were 0.5, then the upper bound distance $d_{[0,1]}(Q, C)$ cannot be greater than 1. This observation is also relevant if we were to apply the equivalent triangle inequality for similarity values. Assuming that,

$$sim(X, Y) = 1 - d_{[0,1]}(X, Y)$$

we can combine the above three equations to give us the bounds on similarity values as,

$$sim(Q, C) \leq sim(Q, T) - sim(T, C) + 1 \text{ (upper bound)}$$

$$sim(Q, C) \geq sim(Q, T) + sim(T, C) - 1 \text{ (lower bound)}$$

This is consistent with the definition of triangle inequality for similarity (Burkhard and Richter, 2000). A similarity measure fulfils the triangle inequality if

$$sim(x, y) + sim(y, z) \leq 1 + sim(x, z)$$

holds for all x, y, z . However, note that if $sim(Q, T) = 0.9$ and $sim(T, C) = 0.1$ then the upper bound on $sim(Q, C)$ becomes 1.8. Observe that this says that Q is quite similar to T , and T is not similar to C .

Therefore, one would expect that C will not be very similar to Q . The upper bound then is only notional. Likewise, if the $sim(Q, T) = 0.1$ and $sim(T, C) = 0.1$, then the lower bound on $sim(Q, C)$ becomes -0.8 . This suggests that the bounds be confined to the range $[0,1]$ ¹¹.

$$sim(Q, C) \leq \min(1, sim(Q, T) - sim(T, C) + 1) \text{ (upper bound)}$$

$$sim(Q, C) \geq \max(0, sim(Q, T) + sim(T, C) - 1) \text{ (lower bound)}$$

We illustrate the behaviour of *Fish and Shrink* algorithm with a hypothetical example. In Figure 15.26 below, we plot the bounds on the similarity values of each case with the query. When a query Q arrives, the *Fish and Shrink* algorithm begins by initializing the lower bound on similarity of all cases to 0, and the upper bounds to 1. The possible range of similarity values is depicted by shaded vertical bands for each case.

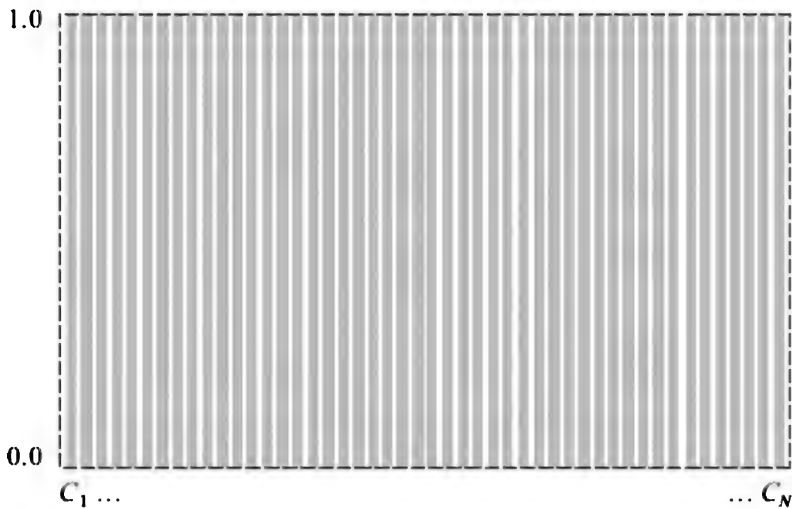


FIGURE 15.26 The *Fish and Shrink* algorithm begins by keeping the widest bounds on the similarity values of all N cases. The feasible values are shown by the shaded vertical band for each case. The lower bound is 0 and the upper bound 1.

It then *fishes* for a candidate, and computes the similarity of the query with the candidate. Having computed this similarity, it then inspects all relevant cases and shrinks their bounds based on the new information and the triangle inequality.

This process continues till the time when there is enough information to select the cases as per the retrieval task. In the figure, it is assumed that the task is to retrieve all the cases above a given threshold t , marked as a horizontal line. Once this line separates the cases with higher similarity then retrieval can stop. If the requirement is to retrieve a set of cases that are sorted in addition then the process of computing similarity will continue till the retrieved cases do not have overlapping bounds. If it is KNN then it will terminate when the lower bounds of K best cases have no overlap with the other cases.

At each stage, the candidate to be chosen is the one that has not yet made it but is most likely to make it into the retrieval set. This is indicated by the precision line p in the figure that marks the upper bound of the case. In the case of threshold based retrieval, the precision line is the highest upper bound of a case whose lower bound is below the threshold limit. In the case of KNN retrieval, it is the highest upper bound of a case that has K or more cases with upper bounds greater than its lower bound.

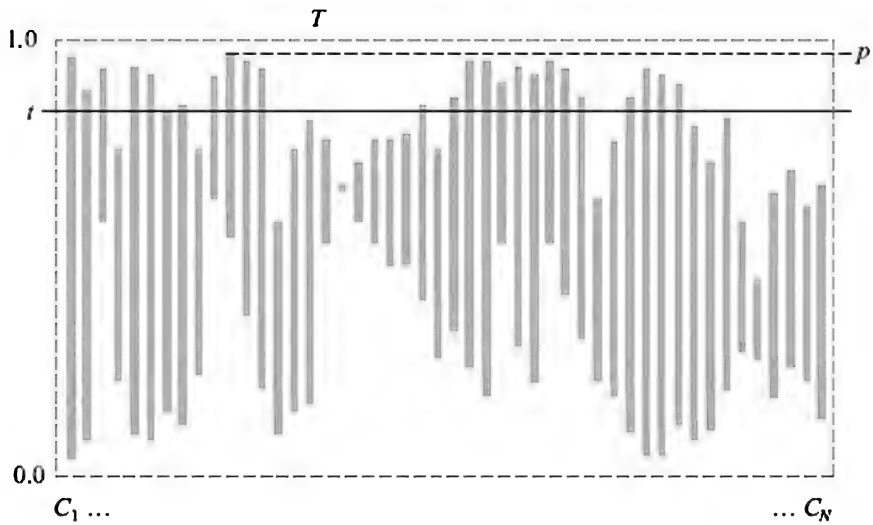


FIGURE 15.27 After computing the similarity $\text{sim}(Q, T_1)$, it shrinks the bounds of all cases based on their similarity with T_1 . The next case to be picked for similarity computation is marked by the precision line.

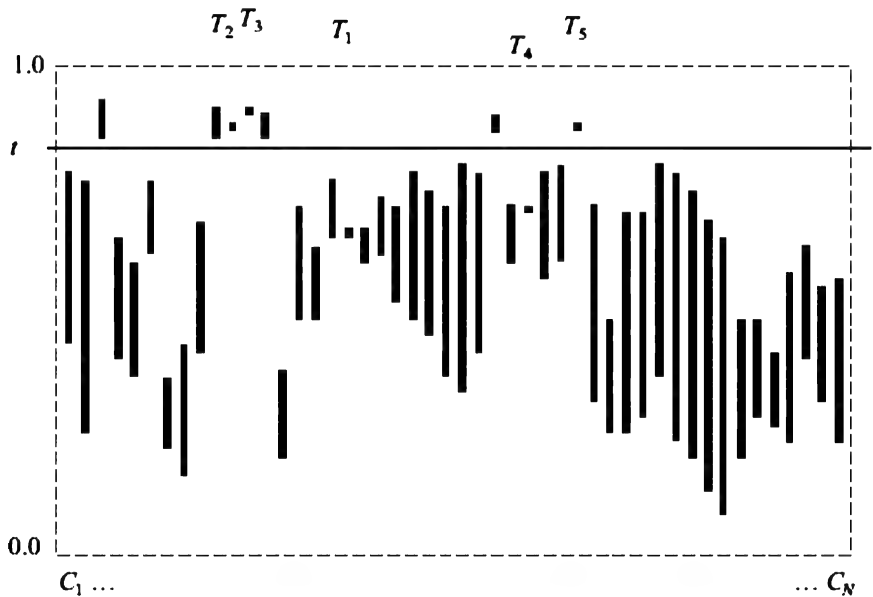


FIGURE 15.28 For threshold-based retrieval, termination happens when no case has bounds that contain the threshold value. The cases inspected are listed on the top. Observe that their similarity values are known completely and they are shown as points.

The algorithm for *Fish and Shrink* retrieval is given in Figure 15.29 below. The version given here is for the case where the task is to retrieve all cases above a threshold T . The algorithm takes as input the following,

- A query Q
- A case base CB

- A pre-computed $N \times N$ similarity matrix SIM . An entry $SIM(i, j)$ stores the similarity between case_{*i*} and case_{*j*}
- A threshold value T

The algorithm uses an array $lower(N)$ to store the lower bounds on similarity of cases with the query Q , and an array $upper(N)$ to store the upper bounds. We assume a similarity function $Sim(case, query)$ is available to compute the similarity value. The variable “Test” stores the index of the case for which the similarity is to be computed. Observe that in the first iteration of the While loop, the first case will always be the one chosen for comparison. We assume a function $GetCase(case\ base, i)$ that fetches the i^{th} case from the case base. The algorithm does not rank the retrieved cases. That is left as an exercise for the reader. The array $simA(N, N)$ stores the precomputed similarities between the N cases.

```

Fish-and-Shrink-T( $q : query, CB : case\ base, simA(n, n), T : threshold$ )
1  for  $i \leftarrow 1$  to  $n$ 
2    do                                     /* Initialize */
3       $lower(i) \leftarrow 0$ 
4       $upper(i) \leftarrow 1$ 
5  done  $\leftarrow FALSE$ 
6  test  $\leftarrow 0$ 
7  while not done
8    do done  $\leftarrow TRUE$ 
9    pLine  $\leftarrow 0$ 
10   for  $i \leftarrow 1$  to  $n$ 
11     do if  $lower(i) < T$ 
12       then if  $upper(i) > T$ 
13         then done  $\leftarrow FALSE$ 
14         if  $upper(i) > pLine$ 
15           then pLine  $\leftarrow upper(i)$ 
16           test  $\leftarrow i$ 
17    $s \leftarrow Sim(GetCase(CB, test), q)$       /* fish */
18    $upper(test) \leftarrow s$ 
19    $lower(test) \leftarrow s$ 
20   for  $i \leftarrow 1$  to  $n$ 
21     do if  $upper(i) \neq lower(i)$ 
22       then                                     /* shrink */
23          $lower(i) \leftarrow Max(lower(i), (s + simA(test, i) - 1))$ 
24          $upper(i) \leftarrow Min(upper(i), (s - simA(test, i) + 1))$ 
25   $R \leftarrow \{\}$ 
26  for  $i \leftarrow 1$  to  $n$ 
27    do if  $lower(i) > T$ 
28      then  $R \leftarrow R \cup \{GetCase(CB, i)\}$ 
29  return  $R$ 

```

FIGURE 15.29 The *Fish and Shrink* algorithm to retrieve cases above a given threshold T . While there are cases in contention, it picks (fishes) the best eligible case for computing similarity. It then updates (shrinks) the similarity bounds of all other cases.

The complexity of the algorithm is quite high. One expects the number of similarity computations to be of order K , where K is the retrieval set size, irrespective of what criteria is used. This is because the best looking case is chosen in each cycle for similarity computation, and the bounds on the similarity value of each case become tighter in each cycle. However, after each similarity computation, all the N cases have to have their bounds

updated. If each similarity computation costs C_{sim} and each update costs C_{update} , then the average case complexity of retrieval is,

$$T = O(K * C_{\text{sim}}) + O(K * N * C_{\text{update}})$$

Observe that the two costs C_{sim} and C_{update} are constant, so the purist will say that the complexity is $O(K * N)$. But we would like to emphasize the fact that the algorithm should be used only when C_{sim} is much larger than C_{update} . Then, if K is significantly smaller than N , it makes sense to highlight the fact that the similarity computation is done only order K times, even though all cases have their bounds updated repeatedly.

15.2.7 Case Retrieval Nets

The algorithms seen so far have a backward chaining or goal directed flavour. Given the query Q , they search all the cases looking for the ones that best match the query. The *Case Retrieval Net* (Burkhard, 1998), (Lenz 1996, 1999), (Lenz and Burkhard, 1996) we study next is a structure that has a forwarding chaining flavour. The *Case Retrieval Net* (CRN) is a two stage, feed-forward network. It falls in the class of *activation spreading* algorithms (Anderson, 1983), (Hendler, 1988). The basic idea is that the query activates certain nodes in a network, followed by a process of activation spreading to neighbouring nodes, eventually activating nodes representing (matching) cases.

The *Case Retrieval Net* is a directed graph that is made up of two kinds of nodes.

1. Information Entity (IE) Nodes Information entity nodes are the *components* used to describe cases. Each *IE* can be viewed as an $\langle \text{Attribute}, \text{Value} \rangle$ pair. For every such pair that exists in the case base, a node is created in the *CRN*. *IE* nodes have an activation value in the range $[0,1]$ which represents the local similarity of the value of the attribute in the query with the *IE* node.

2. Case Nodes A case node represents a case. It may have an activation value in the range $[0,1]$. The value represents similarity of the case with the query, or relevance of the case for the query.

The arcs between nodes are used to propagate activation values. An arc between two *IE* nodes captures local similarity between the two values. Presumably, the two values correspond to the same attribute, but that is not a strict requirement. This local similarity value can be represented by an adjacency matrix defined as follows. Let \mathbf{E} be the set of *IE* nodes and \mathbf{C} be the set of case nodes. Then, local similarity between *IE* nodes is a function $\sigma: \mathbf{E} \times \mathbf{E} \rightarrow [0,1]$, and is represented by the adjacency matrix \mathbf{Sim} . Each nonzero value σ_{ik} in \mathbf{Sim} represents the similarity between the *IE* nodes e_i and e_k and is the weight of the arc between the two nodes. An arc between two *IE* nodes is used to induce activation from one node to another. The *IE* nodes that match the query are assigned an activation

value 1 to start with. The other kind of arc is the *relevance arc* that links an *IE* node to a case node. It signifies the *relevance* of the *IE* node for the case and is a function $\rho: \mathbf{E} \times \mathbf{C} \rightarrow [0,1]$ stored in the adjacency matrix **Rel**.

The following figure illustrates a *CRN* for a meal classification task. Only the arcs with nonzero weights are shown. The weights of the relevance arcs are assumed to be one. The weights of the similarity arcs σ_{ik} represent the author's perception of similarity.

In the above figure, each meal type is assumed to be made up for four¹² components that represent a typical meal. A north Indian meal, for example, is defined as the following case—(Pulav, Phulka, Daal, Lassi). A query case containing all these elements will match the case (class) perfectly. But even a meal (Pulav, Parantha, Daal, Curd) will get a high score.

The process of retrieval in *CRN* is more like the process of reconstruction of a structure from its components or the process of case completion by spreading activation. The high level retrieval algorithm *CRN-T* for finding cases above a threshold, is given in Figure 15.31. First, the *IE* nodes matching the query are activated. The propagation of activation happens in two stages. In the first stage, activation is spread to other *IE* nodes in proportion to the local similarities. Note that this happens in only one step in the algorithm below. This means only nodes directly connected to the initial *IE* nodes will receive activation. In the above example, a query containing “Lassi” will not activate the node “Buttermilk”. One could extend activation spreading to happen in a loop till the values stabilize, like in a Hopfield net. Or one might look for a mechanism to fill in the arc weights accurately by consulting an external source like the *Wordnet* (see Chapter 16).

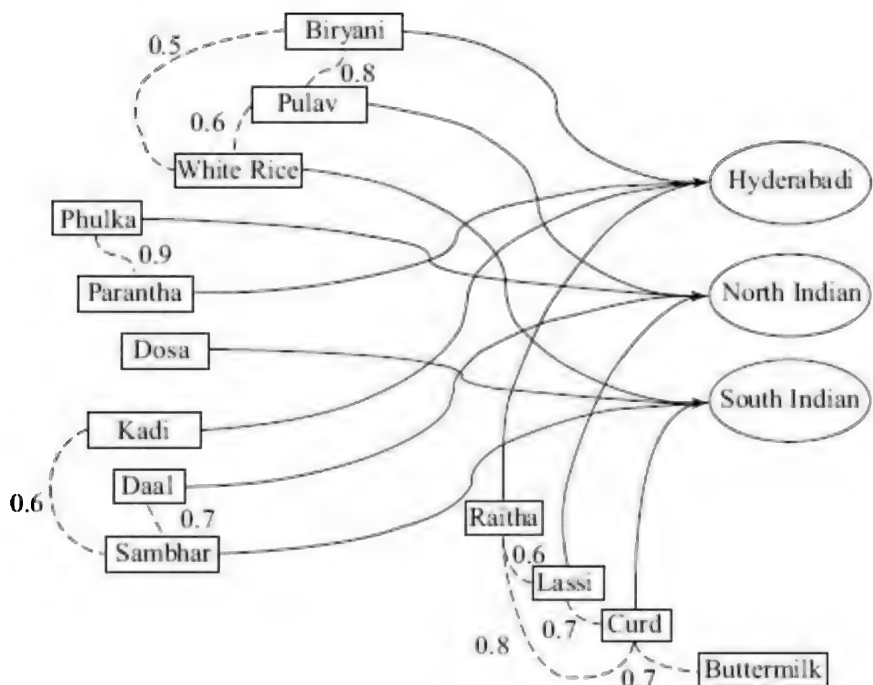


FIGURE 15.30 A sample CRN for a meal classification task. The rectangular nodes are *IE* nodes and the arcs between them represent local similarity. The oval nodes are cases defining a typical meal in that style. The solid arrows capture the relevance of *IE* nodes for case nodes.

In the second stage, activation spreads to the case nodes. The two arrays *E* and *C* contain the activation values of the *IE* nodes and the case nodes respectively. In the algorithm below, we assume the query *Q* to be a list (somehow) containing the indices of the matching *IE* nodes.

The algorithm described above in fact operates in a backward manner. For each *IE* node, the aggregated activation is computed; and then for each case node, the aggregated activation value is computed. The aggregation functions *simAggr* and *relAggr* are of order $O(M)$ steps because activation is transferred from each *IE* node in both. Since M *IE* nodes and N case nodes receive propagated values, the overall complexity of retrieval is $O(M^2 + MN)$ which is $O(M^2)$. This is because it does not exploit the fact that the similarity arcs are likely to be sparse. A variation that only does forward propagation like the AC3 algorithm (see Chapter 9) is likely to be much faster. The reader is encouraged to devise an algorithm that propagates only non-zero values.

15.2.8 Diversity Conscious Retrieval

The basic idea behind CBR is that given a problem, one retrieves similar cases from the case base. This is based on the premise that similar problems have similar solutions. The idea of retrieving K cases instead of

one is that a solution constructed from the K cases is likely to be more robust, specially in ill understood domains.

In recommender systems however, the task is to help a user select a product that best matches her requirement. The retrieval set is not used to construct one solution. Rather it offers a set of choices to the user. It then makes sense that the set of choices is diverse, so that the user has different products to choose from. Diversity, as defined in Section 15.1.2, is a measure of dissimilarity in the retrieval set.

```

CRN-T( $q, E[m], C[n], Sim[m,m], Rel[m, n], T$ )
1  /*  $q$ : query,  $E[m]$ : IE,  $C[n]$ : cases,  $Sim[m,m]$ : IE-sim,  $Rel[m,n]$ :
                                     relevance,  $T$ : threshold */
2  for  $i \leftarrow 1$  to  $m$ 
3      do  $E[i] \leftarrow 0$  /* Initialize */
4  for  $i \leftarrow 1$  to  $n$ 
5      do  $C[i] \leftarrow 0$  /* Initialize */
6  while  $Q \neq ()$ 
7      do /* Query instantiation */
8           $i \leftarrow First(q)$ 
9           $E[i] \leftarrow 1$ 
10          $q \leftarrow Rest(q)$ 
11          $i \leftarrow i + 1$ 
12  for  $i \leftarrow 1$  to  $m$ 
13      do  $E[i] \leftarrow SimAggr(E, Sim)$  /* propagate similarity */
14  for  $i \leftarrow 1$  to  $n$ 
15      do  $C[i] \leftarrow RelAggr(E, Rel)$ 
16   $R \leftarrow \{ \}$  /* Retrieval set */
17  for  $i \leftarrow 1$  to  $n$ 
18      do if  $C(i) > T$ 
19          then  $R \leftarrow R \cup \{GetCase(CB, i)\}$ 
20  return  $R$ 

```

FIGURE 15.31 The CRN-T algorithm begins by activating some IE nodes that the query values. It then uses the function $SimAggr(E, Sim)$ to use the similarity (arc) weights stored in array **Sim** to determine the activation of all IE nodes after propagation. After all IE nodes have been activated, it uses the $RelAggr(E, Rel)$ function to determine the activation of case nodes stored in array **C**.

Diversity conscious retrieval has requirements that may conflict with each other. On the one hand, a retrieved case should be similar to the query. On the other hand, it must be dissimilar to the other cases in the retrieval set. One needs to ensure that in the quest for diversity, there is no significant loss of similarity. Since the retrieval set is constructed incrementally, one needs to use the notion of *relative diversity* of a candidate with respect to the retrieval set being constructed. From the viewpoint of increasing diversity, the next case to be added to the set is the one with the highest relative diversity. We repeat the definition of relative diversity and quality below.

$$\begin{aligned}
 RelDiversity(c, R) &= 1 \text{ if } R = \{ \} \\
 &= \frac{\sum_{i=1, m} (1 - sim(d, d_i))}{m}, \text{ otherwise}
 \end{aligned}$$

Here, m is the number of cases in R at that moment. Adding cases based only on the basis of relative diversity may retrieve cases that are not

similar to the query. Hence, a measure named *Quality* that combines the two aspects of relative diversity and similarity with the query may be used.

$$\text{Quality}(c, R) = \alpha * \text{sim}(d, d_{\text{Query}}) + (1 - \alpha) * \text{RelDiversity}(c, R)$$

Let the *standard retrieval set* (SRS) based on maximum similarity be called R_{SRS} . Let the retrieval set, generated by an algorithm that also considers diversity, be R_D . Then the loss of similarity due to diversity being taken into account is,

$$\text{Loss}_D = \sum_{i=1..K} \text{sim}(Q, C_{\text{SRSi}}) / K - \sum_{i=1..K} \text{sim}(Q, C_{Di}) / K$$

Loss_D signifies the decrease in average similarity, due to diversity being taken into account, and can be controlled by the parameter α in the definition of *Quality*.

The fact that relative diversity depends upon the partially constructed retrieval set means that one cannot choose the entire set in one pass. Instead, one can only choose one element in each pass. This element will in turn influence the choice of the next element to be selected. This means that one will have to make K passes over the case base to retrieve K cases. Since we anyway want to include (high) similarity as a retrieval criterion, the diversity criterion need only be applied to a subset of high similarity cases. That is, the K passes need be done only on the set of high similarity cases. Two algorithms have been reported in the literature.

The *Bounded Greedy* (BG) algorithm (Bradley and Smyth, 2001), (Smyth and McClave, 2001) first selects a subset based on similarity, and then from this subset chooses cases based on diversity. The algorithm is given in Figure 15.32.

```

Diversity-BG( $q$  : query,  $CB$  : case base,  $k$  : retrieval size,  $b$  : bound)
1  $CB_{subset} \leftarrow \text{Sequential-KNN}(q, CB, b*k, n)$  /* most similar bk cases */
2  $R \leftarrow \text{MaxD}(\text{List}(\text{First}(CB_{subset}), \text{Rest}(CB_{subset})), k)$ 
3 return  $R$ 

MaxD( $casesIN$  : cases in,  $setC$  : candidate cases,  $k$  : retrieval size)
1  $R \leftarrow casesIN$ 
2 while  $|R| < k$ 
3   do  $c_{best} \leftarrow \text{GetBest}(setC, R)$ 
4    $R \leftarrow \text{Cons}(c_{best}, R)$  /* add at head of list */
5    $setC \leftarrow \text{Remove}(c_{best}, setC)$ 
6 return Reverse( $R$ )

GetBest( $setC$  : case set,  $R$  : retrieval set)
1  $c_{best} \leftarrow \text{First}(setC)$ 
2  $d_{max} \leftarrow \text{RelativeDiversity}(c_{best}, R)$ 
3  $setC \leftarrow \text{Rest}(setC)$ 
4 while  $setC \neq ()$ 
5   do  $nextCase \leftarrow \text{First}(setC)$ 
6    $setC \leftarrow \text{Rest}(setC)$ 
7    $relativeD \leftarrow \text{RelativeDiversity}(nextCase, R)$ 
8   if  $relativeD > d_{max}$ 
9     then  $c_{best} \leftarrow nextCase$ 
10     $d_{max} \leftarrow relativeD$ 
11 return  $c_{best}$ 

Remove( $c$  : case,  $setC$  : set of cases)
1 if  $c = \text{First}(setC)$ 
2   then return  $\text{Rest}(setC)$ 
3 else return  $\text{Remove}(c, \text{Rest}(setC))$ 

```

FIGURE 15.32 The *Bounded Greedy* algorithm first selects B times the required number of cases based on similarity, and then from this set chooses the K required cases to maximize diversity. The function *MaxD*, takes the first element from the bigger set and augments the set, incrementally adding cases with the highest relative diversity. We assume the function *relative-diversity* that computes the relative diversity of a case w.r.t. a set, as per the formula described in the text.

The *Diversity Conscious Retrieval* (DCR) algorithm (McSherry, 2002) applies finer control on which diverse cases are considered. It employs a control parameter α that controls the loss of similarity for *any* retrieved case. It divides the similarity interval $[0,1]$ into segments of length α and allows diversification of the retrieval set only in cases belonging to *one* such interval. Thus, the maximum loss of similarity for choosing a “bad” case is α . The interval is the one in which the K^{th} similar case in the SRS lies.

```

Diversity-DCR(q : query, CB : case base, k : retrieval size, alfa :
parameter)
1 CBsubset ← Sequential-KNN(q, CB, b*k, n)
2 simLast ← SimK(CBsubset, k)           /* similarity of  $K^{th}$  case */
3 upperBound ← 1                         /* UB of alfa interval */
4 lowerBound ← 1 - alfa                 /* LB of alfa interval */
5 while simLast < lowerBound
6   do upperBound ← lowerbound
7     lowerbound ← upperbound - alfa
8     /* retrieve all cases with similarity higher than
9       lowerBound */
10  R ← CandidateCases(CBsubset, lowerBound)
11  casesIN ← ()                          /* the ones that are definitely in */
12  casesAlfa ← ()                       /* the ones available for diversification */
13  while R ≠ ()
14    do nextCase ← First(R)
15       R ← Rest(R)
16       sim ← Second(nextCase)
17       if sim ≥ upperBound
18         then casesIN ← Cons(nextCase, casesIN)
19         else casesAlfa ← Cons(nextCase, casesAlfa)
20  R ← MaxD(casesIN, casesAlfa, k)
21  return R

```

FIGURE 15.33 The Diversity Conscious Retrieval algorithm first retrieves the bK most similar cases. Function *SimK* should find the similarity of the K^{th} case. It then determines the bounds [*lowerBound*, *upperBound*] of the *alfa* interval in which the K^{th} case lies. Function *CandidateCases* extracts all cases with similarity higher than *lowerBound* from *CBsubset*. Of these, the ones with similarity higher or equal than *upperBound* are definitely in. From the remaining cases, the algorithm *MaxD* from Figure 15.32 chooses the ones that maximize diversity to construct a set of size *k*.

The second algorithm, *Diversity-DCR*, ensures that the most similar cases are always retrieved. In our version of *Diversity-BG*, we have ensured that the most similar case is always retrieved. This is because in function *MaxD*, the first case to be considered always has relative diversity one, because the retrieval set is yet empty. The other cases from *CBsubset* could still be the ones with the lowest similarity values. The *Diversity-BG* does K passes over the set *CBsubset* of size BK , while the *Diversity-DCR* is likely to do a smaller number of passes over the set *casesAlfa*. Apart from that, both algorithms need to do KNN retrieval once. In the algorithms described here, these are done by sequential retrieval, but could be replaced by any suitable faster algorithm.

15.3 Reuse and Adaptation

Case based reasoning works by retrieving the case or cases that are most similar to the query from memory and reusing the solution component for solving the current problem. If the retrieved case matches the given problem perfectly then the retrieved solution could be used directly. However, this may not always be the case¹³. If the retrieved case is a little different from the given problem, then a process of adaptation may be necessary to solve the given problem.

The task of adaptation is a new problem in itself. The input is a possibly flawed solution or lesson. The goal is to produce a good solution. The most

common approach to adaptation has been rule based, though search, or even a case based approach itself has also been tried (Leake, 1996a), (Leake et al., 1996). CBR solves problems by remembering and reusing. Reuse or adaptation however, requires some kind of domain knowledge. The question has been asked that since one has to solve the adaptation problem to solve the original problem, is there any advantage gained by retrieving solutions and adapting them? Why not use the approach being used in adaptation to solve the original problem itself? The answer lies in complexity analysis. Given that most first principles (search based) methods are of exponential complexity, adaptation may involve tinkering only a part of a solution, and may thus be less complex. A formal analysis of this was given by Au, Munoz-Avila and Nau in their paper on case based planning (Au et al., 2002).

Another reason why other knowledge based methods may not be suitable for solving the original problem is the difficulty of *knowledge acquisition* faced by rule based developers. An application to design medical drug composition illustrates this point. Working with AstraZeneca on drug design, Susan Crow says *"Although rules can be applied to suggest formulations for new drugs, it is difficult to acquire effective problem solving rules initially, and equally difficult to maintain these rules as different formulation practices evolve and more difficult drugs require to be formulated"* (Crow, 2001). Instead, the project adopts a case based approach, storing the details of the drug and excipients, together with the formulation that was successfully applied, into a case base (Crow et al., 1998). When a tablet for a new drug needs to be designed, a similar case is retrieved and its formulation reused. Where needed, rules are used for adaptation, for example, *"a harder drug may need a softer filler"* (Crow, 2001).

Let us look at the example of path finding using case based reasoning. We could build a system that can find a path the first time using heuristic search, and store it in a case base for future use (see for example (Raman and Khemani, 1998)). When a new problem arises then the system may retrieve a known path, or a set of paths, from its memory, based on the start and the goal coordinates. To solve the new problem, it may only need to connect its start node and the goal nodes to near points on the retrieved path, as shown in Figure 15.34. These two new subproblems are likely to be much smaller in complexity.

A similar kind of reasoning may happen when using public transport. If one had to travel to a village in Himachal Pradesh in India, one might first construct (retrieve) a travel plan to its capital Simla, and then find a bus onwards to the village. Case based reasoning with adaptation may thus be combined with a means ends, analysis mode of reasoning (see also Chapter 7).

Traditionally, researchers have described two forms of adaptation (Riesbeck and Schank, 1989), (Kolodner, 1993).

Structural Adaptation In structural adaptation, the solution of a case is directly modified to obtain the solution for the new problem.

Derivational Adaptation Derivational adaptation may be employed when the case also contains some steps that are used to arrive at the solution. During adaptation, these steps are executed again with parameters from the current problem.

We also distinguish between two kinds of reasoning tasks, Analytic and Synthetic (Plaza and Arcos, 2002).

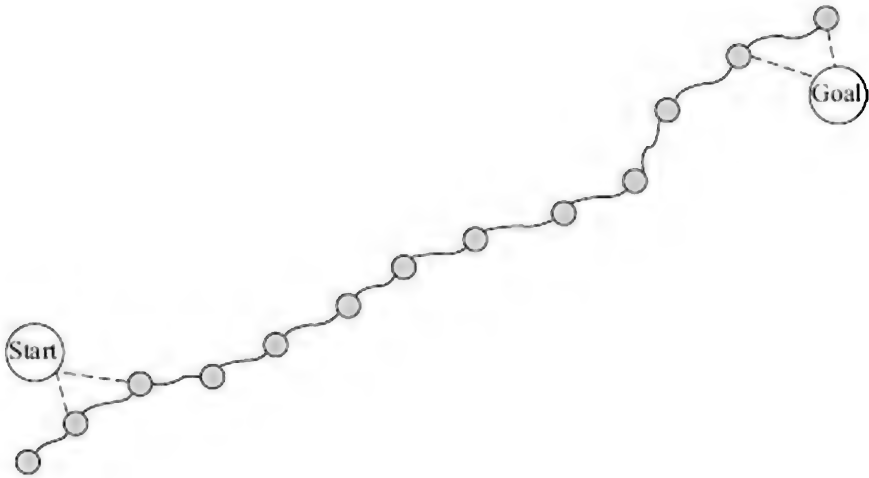


FIGURE 15.34 A multi-modal path finder may retrieve a case shown with the shaded circles. Adaptation may mean finding paths using search to near nodes on the retrieved path.

15.3.1 Analytic Reasoning

Here, the problem is to analyze some problem situation and arrive at some judgment. A typical analytic task is classification.

For example, one may want to classify a person as unhealthy or healthy based on data of age, height and weight. The case base would constitute of a set of data points for which some authoritative source has provided the class label. Given the data of a new person, a CBR system would retrieve the K nearest neighbours and perhaps treat the class label of each retrieved case as a vote for the label of the query case. The class label for the new case could be computed as a simple majority vote. This is also known as *compositional adaptation*. In compositional adaptation, the solution is composed from one or more retrieved solutions. Another way of combining the results would be a weighted majority vote, where the weight of each vote would depend upon the similarity of the query with the case. Cases more similar to the query would have a greater say in deciding the class label.

The same approach that is used to discriminate between healthy and unhealthy persons could also be used to separate spam from genuine emails (see for example (Padmanabhan et al., 2006)). The set of attributes defining the case will be different. For example, the attributes used to describe an email could be a count of the frequencies of certain words,

count of the total number of words, counts of lengths of capital letter sequences, and so on.¹⁴

One may observe that for both these problems, the underlying domain model is not strong enough to define rules to discriminate between the two classes. In the human health problem, given the small set of attributes, one cannot expect a clear cut demarcation boundary between the two classes. In fact, it would not be surprising if two cases (persons) with identical data were to belong to different classes. This is mainly because the three attributes may not be sufficient determinants of health status. Nevertheless, there exist notions like the Body Mass Index (BMI) which are presumed to be health indicators. And BMI is defined on only two, height and weight, of the three attributes described here. Given that we are working with only these attributes, one can still glean some information about the *likely* health status of a person when there are a large number of cases. Then, one could devise the (weighted) voting procedure to give us a value between 0 (say healthy) and 1 (say unhealthy), that could be interpreted as a likelihood of a particular state being true. A value 1 means that the classification outcome for 'unhealthy' is positive.

Likewise, it is not easy to define rules to filter out spam, and one could use the feedback on human selected emails to build a case base with spam and legitimate class labels. Again, one could use a weighted voting procedure to classify emails as spam (positive) or legitimate (negative).

One would have to take care to assign thresholds in a safe manner. In the case of health determination, it might be meaningful to be biased on the side of labelling a case as unhealthy (positive) as opposed to healthy (negative). This is because a *false negative*, labelling someone healthy when they are not, may have serious consequences. A *false positive*, labelling them unhealthy when they are not unhealthy, may only have a price in terms of costs for checkups or unnecessary worry.

The situation is exactly the opposite while labelling spam. A false positive, labelling it spam when it is not, may result in the loss of an important email. A false negative, labelling it legitimate when it is spam, may only add to the reader's irritation.

Typically, in such applications, one tunes some parameters like the threshold that affect the outcome. At one extreme one may label everything negative, leading to a high number of false negatives, and no false positives. At the other extreme, everything is labelled positive leading to a large number of false positives. One may need to choose a threshold carefully after some experimentation, as shown in a *typical* graph in Figure 15.35. The system is a prototype developed to raise and alarm (positive) while monitoring the health of a system. The two plots for false negatives and false positives cross over at some point, suggesting that point as one where the two kinds of errors are equal. Based on the discussion above, however, an application developer may choose a point on either side of the crossover.

Figure 15.35 is a sample plot from experiments on a case based reasoning approach to satellite health monitoring (Penta, 2005). As one can see from the figure, a similarity threshold of 0.995 on the similarity

measure is needed in that system to ensure that there are no false negatives (dismissals), though the number of false positives (alarms) increase at that threshold.

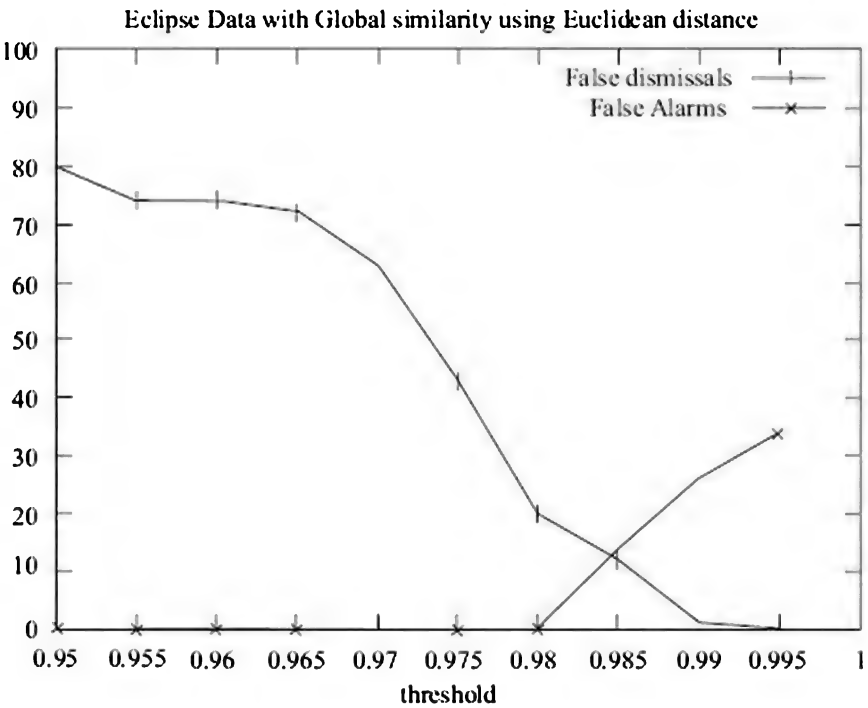


FIGURE 15.35 A plot of false positives (Alarms) and false negatives (dismissals) from a prototype case based satellite health monitoring system (Penta, 2005).

The example of salary prediction we saw earlier is also an analytic problem. Instead of using K neighbours and taking the average salary, one could just take the best matching case and adapt it using rules. One can have a rule like “for every year of increased (decreased) experience, add (subtract) 200 units to (from) the retrieved salary”. Likewise, one can add similar linear interpolation rules for the other attributes. Users of log tables from the twentieth century would have recognized this method!

Derivational adaptation could be used in analytic problems like price estimation. Let us say one is implementing a CBR system in which the price or rental of a house is computed as a function of various features. Then from a case base of houses, one could pick a case with emphasis on location and adapt its price based on the other features, like the number of bedrooms.

15.3.2 Synthetic Reasoning

In a synthetic task, CBR is used to construct a compound artifact like a design, a plan, a layout, or some other kind of a combination of elements.

Adaptation could be of different kinds. It could mean a change in some attribute value, for example the beam width in the design of a house. It could mean substitution of one component with another, for example a metal grill with glass windows. It could also mean addition or deletion of elements in the artefact, thus changing the overall configuration, for example the number of wheels in a car design. If the relation between requirements and a configuration is explicit, one could employ some kind of search to modify the retrieved configuration to suit the requirement. One successful application that did this was the colour matching system *FormTool* developed by General Electric Plastic (Cheetham and Graf, 1997), (Cheetham, 2005). However, this is rarely the case, and it is more common to use a set of hand coded rules to transform or re-instantiate the configuration. An example of this is the system *Wayland* designed to help setup aluminium pressure, die-casting machines (Price and Pegler, 1995). This brings an element of domain specific knowledge into the problem solving task.

The pioneering system *CHEF* developed by Hammoud (1986, 1989) used knowledge represented in *Thematic Organization Packets (TOPs)* for adaptation (see also Chapter 14). *TOPs* embody specialized repair strategies for repairing plans (recipes) with specific kinds of failures. An interesting feature in *CHEF* is that it can run a simulation to detect flaws in a plan. For example, given a task to make a dish containing beef and broccoli, *CHEF* retrieves a recipe for beef and green beans. It first adapts this recipe by reinstantiation of the recipe with broccoli instead of beans. During this process, program critics also suggest that the broccoli be chopped, and adjust the cooking time as well. Now *CHEF* runs a simulation and concludes that the broccoli would become soggy in the given recipe. The *SIDE-EFFECT:DISABLED-CONDITION:CONCURRENT TOP* says that the process of frying beef is generating a thin liquid due to which the broccoli is becoming soggy, and suggests a *SPLIT-AND-REFORM* strategy in which the two are fried separately, thus leaving the broccoli crisp as desired. A key contribution of *CHEF* is that it illustrates *how* a combination of problem solving approaches, using different kinds of knowledge may be necessary for building intelligent systems.

One of the earliest commercial successes of CBR was the system called *CLAVIER* that helped operators at Lockheed configure the layout of ceramic parts to be loaded into an autoclave (Hinkle and Toomey, 1995) (see also (Watson, 1997)). Starting with twenty layouts, *CLAVIER* grew into a system with hundreds of useful cases. The need for adaptation arises when a retrieved layout contains a part that is not on the current agenda for curing. In such a situation, the system needs to substitute it with a similar part. If everything turns out well, the new case is added to the repository. In practice, operators at Lockheed preferred to do the adaptation manually, using the system to check whether the new layout had failed in the past. Manual adaptation was also the chosen route in the *QPAC PCR* system for recording problems in an aluminium foundry into a

case base for troubleshooting (Price et al., 1997). This illustrates the point that even when there is a human in loop solving problems, a CBR system can still learn those solutions and reuse them in the future. This can be a basis for knowledge management in an organizational setting, where the decisions of experienced personnel are captured, and applied even when they have left.

Derivational adaptation has been seen mostly in planning tasks. First introduced by Carbonell (1986), it was employed in various planning systems. *Prodigy/Analogy* (Carbonell and Veloso, 1988) (Veloso and Carbonell, 1993) and *Priar* (Kambhampati and Hendler, 1992) were some of the first systems that used derivational adaptation in a domain independent fashion. The basic idea behind these approaches is that cases are represented at various levels in a hierarchy of abstractions (Cox et al., 2006). When a new problem arrives, a case at an appropriate level is retrieved and refined into a plan using derivational adaptation. *DerSNLP* (Ihrig and Kambhampati, 1997) and *CAPlan/CbC* (Muñoz-Avila and Weberskirch, 1996) applied derivational adaptation to partial order planning.

Constructive adaptation is a variation on derivational adaptation that uses heuristic search to construct the solution, and uses the retrieved cases to guide the search process (Plaza and Arcos, 2002).

We have not discussed the situation when cases are made up of free form text, which is so in many situations where people jot down problems in the form of “tickets” and some experts solve them and jot the solution down. In textual cases, the vocabulary is the words of a natural language. Unlike the structural representation, words are not associated with values from a domain. The case is just a collection of sentences. While words do have a meaning for us, most textual CBR systems do not tend to use meaning or semantics for similarity computation. Instead, we borrow techniques from *Information Retrieval*. *Document Retrieval* is probably a more apt term since the output is usually a matching document. In the *vector space model*, we do assign a value or weight to each word, but that is a statistical property of the word in the context of the document, or even the document collection. We will look at retrieval of text documents in Chapter 16.

15.4 Discussion

Case based reasoning arose from the memory organization work being done at Yale University in the seventies and the eighties (Schank and Riesbeck, 1981; Riesbeck and Schank, 1989; Dyer, 1983). The work done by a bunch of research students under Roger Schank was focused on the kind of representations that would lead to understanding natural language stories that required a considerable amount of situational knowledge (see Chapters 13 and 14). The knowledge was hard coded in structures like Scripts, Goals, Plans, MOPs, TOPS, etc. Many of these were episodic in nature, describing typical situations or typical goal-plan-action connections.

As more and more of these episodic structures were created, the notion of case based reasoning arose. The earlier work focused on using these structures for understanding. This involved a considerable work on the translation from and into natural language, and matching the conceptual dependency structures.

Around the time Hammond wrote his program *CHEF* that could retrieve and adapt recipes and then store them back, the focus shifted to efficient retrieval from a large number of cases. Partly because conceptual knowledge representation for adaptation is a difficult task and partly because the simpler idea of similarity based retrieval found application into many successful industrial projects, the work in CBR gradually drifted into knowledge management like applications.

Current successful applications in CBR employ a uniform case structure along with a similarity measure to determine the best matching cases. Each application stands alone with its own representation schema. The idea of a more “cognitive” agent using memory and learning from experiences has gone onto the backburner. An agent with a diverse memory comprising different kinds of cases of various size and constituents would need a different approach for retrieval. Perhaps the idea of “reminding” in a dynamic memory using conceptual and linguistic cues as indices pointing to the relevant cases will come back. It only awaits further advances in knowledge representation.

15.4.1 Further Reading

Case based reasoning emerged from the work on knowledge structures in memory. A host of programs that employed knowledge structures to understand stories in natural language were being developed. The idea of cases as memories of specific episodes that an agent remembers and reuses, led to the idea of CBR as a memory of past instances. The earliest books that expounded this theme were the books by Roger Schank and Chris Riesbeck (Riesbeck and Schank, 1989), and Janet Kolodner (1993). Chris Hammond's doctoral thesis on *CHEF*, the case based planner explains the notions of revision, adaptation and retaining cases (Hammond, 1986; 1989). A paper by Aamodt and Plaza (1994) appeared just when the simple problem solving approach of remembering and reusing was gaining strength. A collection of essays compiled by David Leake (1996) delved into the theoretical foundations. Two books by Ian Watson (1997, 2002) gathered the vast number of industrial applications on knowledge management. A collection of articles (Lenz et al., 1998) describes some of the advanced topics in CBR. Researchers in German universities devised a software engineering methodology, INRECA, that treated the process as an experience factory (Bergmann et al., 1999). Ralph Bergmann's post-doctoral thesis (Bergmann, 2002) gives a formal grounding to CBR in experience management.



Exercises

1. Refine Figure 15.1 to incorporate implicit forms of knowledge representation.
2. Design a case structure to be used by a real estate site where people can look for renting or buying property. What are the core set of features, and what features could be used in specific domains (like big cities, holiday resorts, mountain or sea locations)?
3. How would one build a dating or a matrimonial site using CBR? Given the profiles of candidate persons, how does one match them to measure compatibility? Would it make sense to define local “similarity” measures that take into account the fact that often “opposites attract”? Or would one transform the given profile (query) to a compatible “desired profile” that could be used for matching the profiles in the database?
4. Find the trigram based similarity, and the cosine similarity between the following sets of sentences,
 - (a) John loves Mary.
 - (b) Mary loves John.
 - (a) Colourless ideas are octagonal.
 - (b) A colourless idea is an octagon.
5. Land on the earth is hierarchically organized into regions. Starting with the continents, there are further divisions into countries, states, districts, etc. Is this hierarchical structure a taxonomy? What is a meaningful interpretation of the distance function?
6. When the case schema is made up entirely of numeric values, then each case can be thought of as a point in N -dimensional space, where N is the number of attributes. Then similarity between two cases could be defined based on some distance measure in the N -dimensional space. Explore the use of Manhattan distance, Euclidean distance, and the Minkowski norm to define similarity functions. What is their correspondence with the aggregation functions defined in this chapter? When will similarity between two cases be zero?
7. The problem in Figure 15.12 involves matching colours, given their RGB values. Experiment with different colour matching functions on your computer, and compare with your visual judgment. Explore other colour representations like the HSV space that represents colours using values for Hue, Saturation and Value.
8. Study the case base in Table 15.3 and try and determine what combinations of factors are associated with a high salary.
9. Consider a case base in which all the attributes are Boolean. Explore the different distance functions that can be devised for two cases in his domain.
10. Modify the KNN algorithm in Figure 15.14 to return (a) all the cases

- with highest similarity, and (b) all cases above a threshold similarity.
11. Using the function *InsertK* defined in Figure 15.23, rewrite the sequential KNN retrieval algorithm.
 12. Can we think of the *kd-tree* as a taxonomy of cases? What are the similarities of *kd-trees* with taxonomies? What are the differences?
 13. The algorithm for building a *kd-tree* in Figure 15.17 does not assign parent pointers at each node. Modify the algorithm to do so. [*Hint:* Assign “NIL” parent to root and others in recursive calls.]
 14. Modify the algorithms *Build-kd-tree* and *KNN-kd-tree* to work with similarity values instead of distance values.
 15. The *KNN-kd-tree* algorithm applies the BOB test whenever it goes to a child node. Modify the algorithm to apply it only on leaf nodes. Which version is likely to run faster?
 16. Specify the termination criteria and the precision line definition when the *Fish and Shrink* algorithm has to retrieve *only* the *best* cases.
 17. Modify the *Fish-and-Shrink-T* algorithm for the KNN retrieval task.
 18. Extend the *Fish and Shrink* algorithm to continue till the cases are sorted in decreasing value of similarity. [*Hint:* continue the *Fish and Shrink* process till the bounds do not overlap.] What set of cases will be used here?
 19. Extend the *Fish and Shrink* algorithm to return the similarity values as well.
 20. Given a query, how would one efficiently generate the list of indices required as input for our *CRN* algorithm in Figure 15.31?
 21. Modify the *CRN-T* algorithm to retrieve the *K* best cases.
 22. The algorithm *CRN-T* requires an *IE* node for every attribute-value pair. These nodes could be constructed from the cases in the case base. However, for numeric attributes, the query may contain values that do not match any *IE* nodes. Extend the *CRN-T* algorithm to activate the best matching *IE* nodes with appropriate values for numeric attributes.
 23. Write the function *relative-diversity(case:C, retrieval set:R)* used in Diversity-BG procedure. Keep in mind that the cases in the list returned by Sequential-KNN, are pairs containing the case-id and its similarity with the query.

¹ Subsequently, this step was expanded to Recycle, Refine and Retain (Göker and Roth-Berghofer, 1999).

² More papers at the sites for Mixed Initiative workshops held during ECCBR02 and ICCBR03 <http://home.earthlink.net/~dwaha/research/meetings/eccbr02-micbrw/>
<http://www.iccbr.org/iccbr03/workshops/dwaha/research/meetings/iccbr03-micbrw/index.html>

³ Thanks to N.S. Narayanaswamy.

⁴ Looking at candidates for boxing matches, however, 65 and 68 might be considered to have low similarity because the “qualitative

interval” that defines Welterweight category is 63.5 kg and 66.7 kg
—<http://en.wikipedia.org/wiki/Welterweight>

5 Metric (mathematics).
http://en.wikipedia.org/wiki/Metric_%28mathematics%29

6 Note that conditions 1 and 2 together produce positive definiteness.

7 I remember a bright four year old asking “But *why* do they call it a *fan*? Why not something *else*?”. She was referring to the ceiling fan, a common fixture in Chennai homes.

8 In practice though, an ENT specialist would also be to tell you that you have a problem with your eyes or ears.

9 We use the same symbol *sim* for both local and global similarity. The arguments to the function determine what we are talking about.

10 In an Implementation of CBR in the manufacturing domain (Khemani et al., 2002), the shop floor personnel gave high weights to all attributes, saying that each was very important. While that may be true from the manufacturing perspective, it need not be the case for measuring similarity between two cases.

11 Thanks to Sutanu Chakraborti for discussing this point.

12 In the interest of brevity.

13 “*You can never step into the same river, for new waters are always flowing on to you.*” -Heraclitus of Ephesus (535–475 bc).

14 See for example, <http://archive.ics.uci.edu/ml/datasets/Spambase>

Natural Language Processing

Sutanu Chakraborti

Chapter 16

In Chapters 13 and 14, we have emphasized the importance of knowledge representation in AI systems. Once a model of the world is captured in formal representations like First Order Logic, we can devise powerful reasoning mechanisms. Humans, however, seem to be seamlessly effective in communicating with each other in natural languages like English. *Natural Language Processing (NLP)*, a subfield of AI, attempts to build computational systems that can converse with us in natural language. NLP has two subdisciplines: *Natural Language Understanding (NLU)* aims at building systems that can make sense of free-form text. *Natural Language Generation (NLG)* aims at building systems that can express their knowledge or explain their behaviour in natural language.

Building systems that understand natural language is both important and challenging. It is important because intelligence is all about making sense of the world around us, and the world is more likely to present itself to an intelligent agent in natural language, than in structured representation languages. Merrill Lynch estimates that more than 85% of all business information exists as unstructured text (Blumberg et al., 2003). Examples of such free-form data abound in the form of emails, memos, notes from call centres and support operations, news, user groups, chats, reports, surveys, white papers, research articles, presentations and Web pages. It will take an astronomically large number of 'man hours' to render them all into representations machines are comfortable with. Even if all of humankind were engaged to burn the midnight oil out to achieve this, the problem is not going to go away, because we are generating unstructured documents at a far higher pace than we can assimilate them. It was estimated in 2006 that more data will be produced in 2007, than has been generated during the entire existence of humankind (Panurgy, 2006). AI systems that rely critically on formal representations are analogous to a well laid out network of pipes and taps within a building, whose overhead tank has to be filled in by battalions of people carrying buckets. In the context of this metaphor, an NLU system can be thought of as a pump that feeds the overhead tank automatically. The growth of the World Wide Web has seen a renewed interest in NLU systems that can facilitate knowledge engineering from a

diverse collection of unstructured documents. There are many other interesting applications of *NLP* systems, which we review in detail in Section 16.3.

Understanding natural language is challenging. It may be illustrative here to compare a natural language like English against a programming language like C. Consider a news report headline:

*“Stolen Painting Found by Tree”*¹

Our first observation is that the heading of the news report is not a well-formed English sentence, yet it makes perfect sense to an average reader. Ideally, an *NLP* system should be robust to such variations. In contrast, a C program with statements that do not conform to the underlying grammar will be rejected by a compiler. Ill-formed constructs are typical in our conversations (A: “*Looks nice, few typos though*”, B: “*Doesn’t matter*”), and more recently in SMS messages (“*C u b4 3*”).

Secondly, an *NLU* system must be able to effectively handle ambiguity. The news headline above has two possible interpretations, though an average reader has no trouble favouring one over the other. For us, it appears obvious that a tree cannot go around searching for a stolen painting. Programming machines to do the obvious, however, turns out to be challenging. In contrast to natural language texts, a computer program typically has a single unambiguous representation, or else a machine would have difficulty executing it.

Thirdly, unlike systems that process and execute programs, understanding natural language often needs recourse to a body of common sense and background knowledge. Let us consider the following example:

“Shruti ordered a pizza. She left a tip before leaving the restaurant.”

To understand the above sentences, the reader must have knowledge of what people typically do when they visit restaurants (see Section 14.5). Similarly, one needs to have knowledge of how a cricket match is played, in order to be able to make sense of a news reporting how Sachin Tendulkar went on to score a century in a World Cup match. Encoding all relevant common sense and background knowledge and incorporating them appropriately in *NLU* systems has proved to be the holy grail of *AI*.

It is clear that we need to model the complex interaction of several phenomena to be able to understand how humans process natural language. In this respect, *NLU* is a scientific activity in a spirit very similar to natural sciences (like physics) where language is a natural artifact being studied, and the goal is to arrive at sophisticated models of language and its understanding. This involves the coming together of several disciplines like cognitive science, theoretical linguistics, psychology, machine learning and artificial intelligence. *NLU* is also an engineering activity, in that we attempt to build computational models that

can make sense of textual data in the limited context of a given task or application. Several real world applications have been built; we will see examples of such applications later in this chapter.

16.1 Classic Problems in NLP and Schools of Thought

While linguists are interested in characterizing languages and processes that account for its effective use, the field of *Computational Linguistics* (which we use interchangeably with *Natural Language Processing*) restricts attention to models that are realizable on a computer.

The number of distinct English sentences is infinite. An idealized computer as conceptualized using a Turing Machine has a finite number of states, but can recognize (or accept) a language having infinite number of strings. On the surface, designing an automaton that can accept all English sentences and reject all invalid ones seems doable. In the light of the problems discussed in the introduction, however, defining the grammar would be incredibly challenging, and getting around ambiguities harder. Even if we did manage to construct models for a subclass of English, we would not be sure we had systems that actually “understood” the sentences, since we have observed that understanding needs a wealth of common sense and background knowledge that is not (yet) available in a form that machines can readily use. It makes sense to systematically analyse the fundamental problems that we need to address, if we were to make some progress in understanding the conceptual basis of natural language. This is the first major step in building machines that can reason with natural languages.

Physics is about matter, but instead of studying all varied forms of matter, it starts out by examining basic building blocks like atoms, what are shared by all matter. In *NLP*, the most fundamental building block is a word. We associate properties with atoms; a sulphur atom is different from a chlorine atom. We associate properties with words; an “elephant” refers to something and a “donkey” to something else. A sulphur atom can be broken down further into particles like electrons, protons and neutrons, but the important point is that a sulphur electron is indistinguishable from a chlorine electron. Similarly, a word can be broken down to letters, but the letter “n” in “elephant” is indistinguishable from the letter “n” in “donkey”. Thus, an atom is a specific configuration of primitives (like electrons, protons, neutrons) that has a unique property, and a word is a specific configuration of primitives (letters) that has a unique property. Let us start our study of language by examining the properties of words.

The first property of a word is that it has a meaning; a word is a surrogate for something in the material or the abstract world. Letters that constitute a word don’t have meanings of their own (though some words are just single letters); so in that sense, a word is the smallest linguistic

element with a meaning. In *NLP*, the study of word meanings is called *lexical semantics*. The word “lexical” is used whenever we want to refer to processing at the level of a word. One central question is: how do we make machines understand the meanings of words? Humans use dictionaries which explain the meanings of complex words in terms of simple ones. For machines to use dictionaries, we have two problems. The first is, how do we communicate the meaning of simple words (like “red” or “sad”)? We have also discussed the problem of defining the semantics of such “words” in *FOL* in Chapter 13. The second is, to understand the meanings of complex words out of simple ones, we would need the machine to understand English in the first place. The first problem has no easy solution; there are words whose meanings are expressed better in the form of images or when contrasted with other words (“orange” versus “yellow”). The second problem of defining words in terms of others can be addressed using a knowledge-representation formalism like a semantic network. The *WordNet* (Miller, 1995) is a massive network of words compiled manually over 10–15 years, where each word is extensively annotated and its inter-relationships to other words are also specified. A fragment of the *WordNet* network is shown in Figure 16.1.

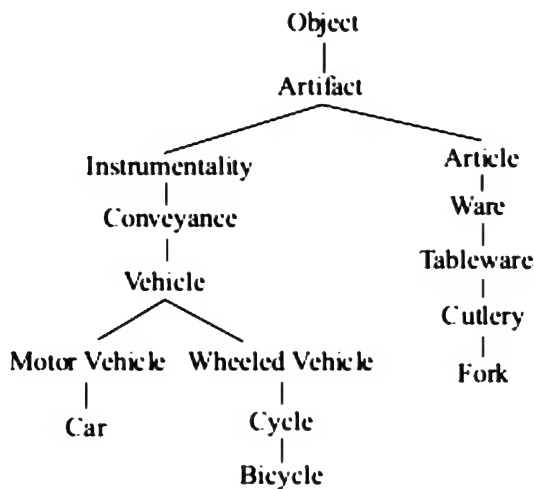


Figure 16.1 A fragment of the *WordNet* Noun Network showing a concept hierarchy (an adapted version of an example in (Jiang et al., 1997)).

In the terminology of *WordNet*, a word has a form (the sequence of letters that comprise it). A word form could have multiple meanings; this is referred to as *polysemy*. The word “bank”, for example, can refer to a financial bank or a river bank. The different meanings of a particular word form are often referred to as *senses*. Determining the right sense in which a word form is being used is a nontrivial problem that goes by the name of *Word Sense Disambiguation (WSD)*. *WSD* relies on contextual

information as obtained from neighbouring words to determine the correct meaning of a given word.

It is also commonplace that several word forms map onto the same meaning; this is referred to as *synonymy*. Search engines like Google that aim at retrieving relevant documents based on very few search terms need to handle synonymy and polysemy effectively. If a search engine can recognize synonyms, it can ensure that a prospect document does not get left out. If a search engine handles polysemy effectively, it can prevent an irrelevant document from getting retrieved.

The second property of a word is its *Part of Speech*. The Part of Speech (POS) dictates the suitability of words to tie up with each other to give rise to grammatical sentences. An analogy can be drawn to valency of atoms, which is primarily responsible in dictating which molecules are possible and which are not. A molecule in which atoms don't have their valencies pairing up is like an ungrammatical sentence where the parts of speech of words are not in agreement with each other. There are primarily five Parts of Speech: nouns, prepositions, verbs, adjectives and adverbs. A word can be associated with more than one Part of Speech. As with *WSD*, determining the right Part of Speech of a word requires one to look at neighbouring words; this is referred to as the *Part of Speech Tagging* (POST) problem.

The third property of a word is its *morphology*, which is its structure or form. This refers to the sequence of letters in the words. There are systematic ways in which the form of a *root word* (like "sing") can be changed to give birth to new words (like "singing"). Individual constituents of words (like "sing" and "ing") are sometimes referred to as *morphemes*. *Inflection* is the process by which a new word is obtained from its root, such that the new word preserves the Part of Speech of the root word. In contrast, *derivation* may lead to a new word that has a different Part of Speech. Thus, the transformation *sing*→*singing* is inflection, whereas the transformation *cheer*→*cheerfulness* is derivation. Languages like Sanskrit have a large number of transformation rules which lead to generation of complex words by combining morphemes, most of which have specific roles in dictating the meaning of the resulting word. In contrast, inflection and derivation rules in English are simpler, though there are plenty of exceptions (the past tense of "eat" is "ate" and not "eated"). Later in this chapter, we will see that Finite State Machines are useful in modelling morphological operations in English. Search engines need morphology tools that yield the root forms of words, so that a query "*dogs biting cats*" can retrieve a document titled "*a dog bites a cat*". Another application that word processors and Google very often use is a *spell-checker* which tries to identify the correct word form, given its incorrectly spelt version.

The fourth property of a word is its pronunciation or *phonetics*. This aspect is of particular importance to researchers in speech processing. Since the focus of this chapter is exclusively on processing of written (and not spoken) text, this topic is outside the scope of this chapter.

Words come together to form bigger semantic units which we call sentences. For a sequence of words to form a sentence, they must be arranged in accordance with the rules of grammar. The meaning of a sentence is composed from the meanings of words. There are two important problems in *NLP* relating to this phenomenon of words coming together to form sentences. The first problem is: given a sentence, how do we break it up into chunks in a way that is consistent with the grammar of the language? This is the *parsing problem*. Figure 16.2 shows a sentence and its representation after it has been parsed. The sentence is: “*The idea that machines understand languages fascinates us*”. The second problem is to account for how the meaning of a sentence is *composed* from the meanings of the words in it. This is often referred to as *compositional semantics* to distinguish it from lexical semantics. As we have noted before, the notion of sentence “meaning” can get quite tricky. True understanding may need a wealth of background and linguistic knowledge, as also knowledge inferred from the context in which a sentence is uttered, and these can interact in complex ways. However, even as full scale understanding is an unsolved problem, there have been interesting techniques and knowledge representation formalisms that have been proposed towards addressing both the problems mentioned above. These will be covered in Section 16.2.

Just as words come together to form sentences, sentences come together to form paragraphs or documents. The technical name for a group of sentences conveying some information is *discourse*. Understanding discourse is a ‘grand challenge umbrella’ problem for *NLP* that encapsulates several long and short term problems. One immediate implication of effective discourse processing is that we could design systems that could read a story and answer questions based on it. This would also mean that we would have automated systems that can process Web pages and construct representations of these Web pages that have richer representations of the underlying semantic content. Web search engines, in turn, could exploit these systems to facilitate a more effective comparison of the queries to documents. While discourse processing is hard, we will consider in this chapter one subproblem which is relatively well studied. It is called *anaphora resolution*; a special case of resolving pronoun references. Let us consider the sentences “*Ram hit Mohan. He was badly hurt.*” and contrast this with “*Ram hit Mohan. He escaped.*” In the first case, the pronoun “he” refers to Mohan, and in the second case to Ram. Anaphora resolution in this example refers to the problem of determining which entity is referred to by “he”.

The idea that machines understand natural languages fascinates us

Part of Speech Tags:

S: Sentence

NP: Noun Phrase

N: Noun

V: Verb

JJ: Adjective

PRON: Pronoun

Grammar Rules:

$S \rightarrow NP V NP$

$NP \rightarrow N$

$NP \rightarrow PRON$

$NP \rightarrow Det N$

$NP \rightarrow NP \text{ that } S$

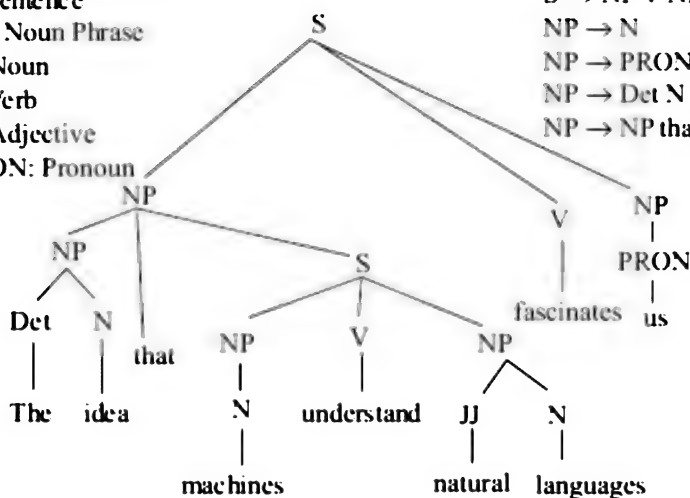


Figure 16.2 A sentence and its structure.

In natural language, there are occasions where the intent of the speaker does not have a straightforward correspondence with the words or sentences she uses. For example, imagine a pedestrian calling out “Taxi!!!” and the taxi driver shouting back “Pedestrian !!!”. This is referred to as the problem of *pragmatics*.

In Section 16.2, we will take a closer look at the central problems in *NLP* identified above. We will focus our attention on some representative approaches aimed at addressing these problems.

Two Schools of Thought

Before we begin our discussion on specific techniques to address *NLP* problems, it may be worth noting that there are principally two schools of thought in *NLP*, namely the *rationalistic* one and the *empirical* one. Till the mid-eighties, *NLP* systems were built using hand coded grammars and symbolic rules. There were two main limitations of this so called rationalistic approach. First, such systems were brittle and often restricted to specific domains. A classic example is the *SHRDLU* system (Winograd, 1971) built by Terry Winograd as part of his PhD at MIT. The system could build a dialog about a world of toy blocks and move blocks around based on commands in a natural language. Secondly, developing these systems needed a lot of manual intervention for knowledge engineering. These limitations were critical bottlenecks in the way of development and field deployment of real world *NLP* systems. Empirical methods in *NLP* were motivated by the need to address these limitations

and gained prominence in the nineties and continues to dominate the bulk of *NLU* work reported today. Empirical *NLU* relies on *statistical machine learning* on a corpus. Given a collection of naturally occurring sentences as input, the aim is to algorithmically acquire useful information about the language. Empirical *NLP* has had much remarkable success in several language tasks such as syntactic and semantic analysis, machine translation, discourse analysis, information extraction and in speech processing, though many believe that it is not appealing from the cognitive standpoint. Noam Chomsky, one of the pioneers of the rationalist school of thought, proclaims that language cannot be learnt from data, and that most of language is innate to humans. This line of thinking has been popularized more recently by Steven Pinker, who in his book *The Language Instinct* maintains that humans do not learn any more language from corpuses, than birds learn flying by statistically analysing patterns of flights of other birds.

16.2 Basic NLP Techniques

16.2.1 A Spelling Checker

Perhaps the most common application that we can think about at a word morphology level is that of detecting and correcting spelling errors. Once a spelling error is detected using a standard dictionary, the system needs to suggest candidate corrections. A simplistic strategy would be to just compare the number of letters in common between the misspelt word and each of the candidates. However, the limitation of this approach is that it fails to take into account the order of letters. Thus, 'WRONG' and 'GONER' are treated as equally likely replacements for the misspelt word 'GONIR'. This limitation can be overcome by considering sequences of two letters (bigrams) or sequences of three letters (trigrams) as basic building blocks of a word. For example, the bigrams and trigrams of the word 'GONER' are {'GO','ON','NE','ER'} and {'GON', 'ONE', 'NER'} respectively. A simple measure of a trigram based similarity between strings *A* and *B* is as follows:

$$sim(A, B) = \frac{|trigram(A) \cap trigram(B)|}{|trigram(A) \cup trigram(B)|}$$

Here, *trigram(S)* refers to the set of trigrams in the string *S*. In the example above,

$$\begin{aligned} \text{trigram}('WRONG') &= \{'WRO', 'RON', 'ONG'\} \\ \text{trigram}('GONER') &= \{'GON', 'ONE', 'NER'\} \\ \text{trigram}('GONIR') &= \{'GON', 'ONI', 'NIR'\} \end{aligned}$$

Thus,

$$\text{sim}('WRONG', 'GONER') = 0$$

$$\text{sim}('GONER', 'GONIR') = 0.20$$

It may be noted that because of their simplicity, n -gram based approaches, of which the bigram and trigram are special cases, have been used in various practical applications that need fast string matching.

Despite being simple and easy to use, the n -gram approaches suffer from the limitation that they are overtly sensitive to insertions, deletions and substitutions. In the example above, the trigram similarity seems to underestimate the similarity between 'GONER' and 'GONIR' which differ by just one letter. Worse still, the trigram similarity between 'GONER' and 'GOBER' is 0. To overcome this limitation, commercial spellcheck tools that come with word-processing software use the notion of *edit distance*, which is explained below.

Let us consider two strings: GREAT and GRATE. Let us attempt to transform GREAT to GRATE by using a combination of three primitive operations: letter insertions, deletions and substitutions. There are several ways in which a word can be transformed into another. For the current example, two such transformations are shown in Figure 16.3.

A cost is assigned to each primitive operation, viz. insertion of letter, deletion of letter and substitution of letter. A total cost of transformation is computed as the sum of each of these primitive costs. Taking the costs to be 1 for each of the three primitive operations, the total costs for the two transformations shown in Figure 16.3 are 2 and 6 respectively. The *edit distance* is defined as the *least* cost of transforming one string to another. In the above example, the edit distance between 'GREAT' and 'GRATE' is 2.



Figure 16.3 Two different ways of transforming one string to another.

The process of enumerating all possible transformations from one string to another and choosing the least-cost transformation is computationally expensive. Fortunately, dynamic programming comes to

the rescue by cleverly avoiding certain redundant computations. The details of this algorithm are presented in (Dasgupta et al., 2006).

There is yet another interesting way of addressing the spell-check problem. Consider a wrongly spelt word W and words A , B , C that are candidate corrections for W . Let $P(A|W)$ be the posterior probability that A is the correct replacement for W . By using Bayes' rule of probability, this is a posterior term which is proportional to the product of two quantities: the likelihood term $P(W|A)$ and the prior term $P(A)$. Similarly, the likelihood and prior terms are computed for B and C as well. The word with highest posterior probability is chosen as the replacement for W . While the likelihood term takes care of systematic processes (like keyboard proximity of letters) that lead to a typo, the prior term ensures that a more frequent word is favoured over a less frequent one. The prior probabilities can be estimated using a standard corpus. (Kernighan et al., 1990) describes an approach to estimate the likelihood by recording frequencies of replacement of each English letter by another.

16.2.2 Morphology using Finite State Transducer

In a way not very different from how words act as building blocks for sentences, words themselves are built up from a sequence of *morphemes*. Computational morphology encompasses two areas. The first is *analysis*, where a word is broken into its constituent morphemes. The second is *synthesis*, wherein a word is composed from morphemes. While it may appear that "parsing" a word to its morphemes is a simpler task than parsing a sentence, there are still interesting challenges. Consider the ambiguity in parsing the word "foxes" (a plural noun or a singular verb) for example, which cannot be resolved without access to contextual information. Three different sources of knowledge are necessary for morphological processing. The first is the *lexicon* which has a listing of words and morphemes along with their roles (for example, adding an *s* gives the plural form of a noun). The second is *morphotactics*, which is a set of rules that govern which endings go with which words (for example, 'er' can follow 'do' but not 'be'). The third is a set of rules to allow for change in spelling (for example, 'fly + s' should give rise to 'flies' and not 'flys').

The apparatus needed for morphological processing are concatenation and a mechanism to allow certain combinations and rule out some others based on the characters processed. These can be conveniently captured using a *Finite State Automaton* (FSA). An FSA makes transition between states driven by a sequence of input symbols. One or more of these states are marked as acceptors, and the automaton is said to have accepted or recognized the string if it ends up being in one of the acceptor states, after all symbols are processed. An extension of this basic idea is that of *Finite State Transducers* (FST), which output symbols as it makes state transitions. Thus, an FST can be viewed as a

mechanism of taking in an input string and generating an output string, and this is ideally suited to the task of morphological processing.

The development of morphological processors is often modularized in practice, by adopting the notion of a two-level morphology. The first module takes in a surface form of a word (say “birds”) and splits it into possible morphemes (“bird” + “s”). The second module converts these morphemes to root forms and tags identifying their morphological features (“bird” + *N* + plural).

Morphological processing is an essential prerequisite for *NLP* parsers and Part of Speech taggers, and also for applications like machine translation, question answering and grammar checking. While tasks like Information Retrieval also need a knowledge of morphology in principle, they tend to do away with elaborate morphological processing by using simpler and faster algorithms for arriving at canonical representations for variants of a given word. A popular tool is the Porter’s algorithm (Porter, 1980).

16.2.3 Lexical Semantics using WordNet

WordNet (Miller, 1995) is a lexical reference system developed at Princeton University. It is based on psycholinguistic theories of human lexical memory. While at a gross level, it can be viewed as an electronic thesaurus, a critical distinction is that *WordNet* organizes lexical information in terms of word meanings rather than word forms. The lexicon is divided into five categories: nouns, verbs, adjectives, adverbs and function words. Words from various Parts of Speech are organized into synonym sets (also called *synsets*), each representing one underlying lexical concept. Different relations link synsets with other synsets. The following are examples of *WordNet* relations that are defined over nouns. *Synonymy* refers to a similarity of meaning. *Antonymy* is a lexical relation between word forms, not a semantic relation between word meanings. Thus, *rise* is an antonym of *fall*, but is not an antonym of *descend*. *Hyponymy* and *Hypernymy* are semantic relations between word meanings. A concept represented by a synset is said to be a hyponym of a concept represented by a different synset, if the former “is-a(kind of)” the latter. The latter would be called a hypernym of the former. For example, {*maple*} is a hyponym of {*tree*}, and {*tree*} is a hyponym of {*plant*}. *Meronymy* and *holonymy* are again semantic relations between word meanings. A concept represented by a synset is said to be a meronym of a concept represented by a different synset, if the former “is a part of” the latter. The latter would be called a holonym of the former. *WordNet* also has Morphological Relations which are lexical in nature. For example, {*trees*} is morphologically related to {*tree*}; and thus we can move from {*trees*} to {*tree*} by suffix stripping. More involved techniques may be required to handle all inflections and derivations.

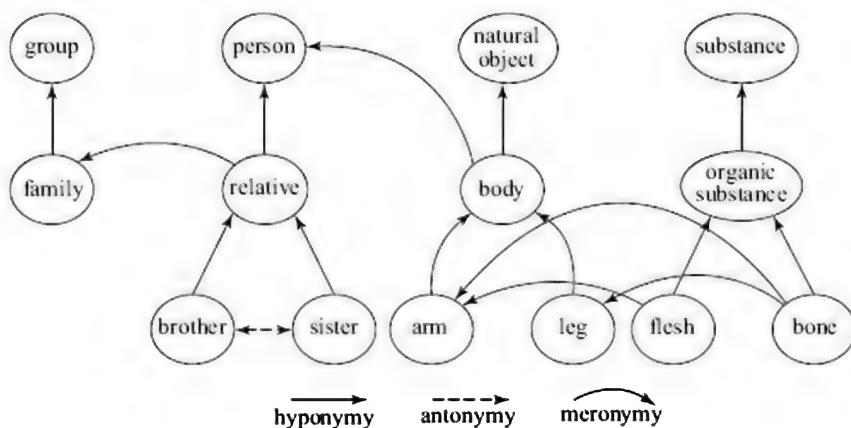


Figure 16.4 *WordNet* relations. <http://wordnetcode.princeton.edu/5papers.pdf>

An interesting application of *WordNet* is that it can be exploited to compute a numeric estimate of semantic relatedness between synsets in the closed interval [0,1]. There is an open source Perl utility *WordNet::similarity*² (Pederson et al., 2004) that provides access to several different measures based on the graph theoretic structure of *WordNet*. We discuss a simple example below to illustrate one such measure.

Let us consider a hypernymy tree in *WordNet* as shown in Figure 16.5. Here, the hypernym tree with the smallest path from the token 'fox' to the root node 'entity' is shown. The similarity between any two nodes in the tree is computed by the simple formula:

$$\text{Similarity}(\text{node}_1, \text{node}_2) = 100 - 100 * (\text{Distance}_{\text{LCS}} / \text{Distance}_{\text{ROOT}})$$

where

$\text{Distance}_{\text{ROOT}}$ = Sum of distances of the two nodes to the root

$\text{Distance}_{\text{LCS}}$ = Sum of distances of the two nodes to the Least Common Subsumer (LCS)

The LCS is defined as the most specific concept that is the shared ancestor of the two nodes.

In the example shown in Figure 16.5, the similarity between the token 'fox' and 'wrongdoer' is calculated below. The LCS here is 'wrongdoer'.

Distance of the node 'fox' from the node 'wrongdoer' is 2.

Distance of the node 'wrongdoer' from the node 'wrongdoer' is 0

Therefore $\text{Distance}_{\text{LCS}} = 2 + 0 = 2$

Distance of the node 'fox' from the root 'entity' is 6

Distance of the node 'wrongdoer' from the node 'entity' is 4

Therefore $\text{Distance}_{\text{ROOT}} = 6 + 4 = 10$

$\text{Similarity}(\text{'fox', 'wrongdoer'}) = 100 - 100 * (2 / 10) = 80\%$

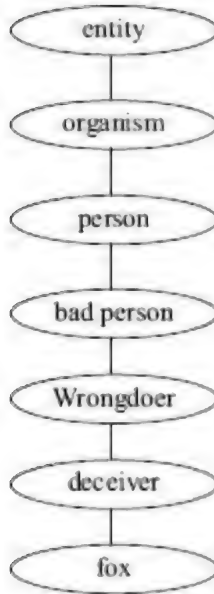


Figure 16.5 A *WordNet* hierarchy.

16.2.4 Word Sense Disambiguation

WordNet records the different senses in which a word can potentially be used. Since each synset corresponds to a distinct meaning, each sense of a word maps to a distinct synset. Word Sense Disambiguation (WSD) involves identifying the sense of a word in a given piece of text, by making use of other words in context and grammatical cues. State-of-the-art WSD systems rely on Machine Learning techniques to establish the mapping.

Supervised WSD systems rely on annotated corpora, in which each polysemous word is manually labelled with its correct sense. An example of one such freely available corpus is *Semcor*. A classifier such as a neural network or an instance based learner (refer Chapter 18) is trained on such a labelled corpus. The learner uses the training data to establish a mapping from the contextual features of a word, such as words in its proximity, to the correct sense of the word. A new word instance is viewed as a test case and assigned a sense based on the predictions of the trained model.

A major bottleneck with supervised WSD systems is the reliance on

hand-annotated corpora. Unsupervised WSD systems overcome this limitation. Each word instance is described in terms of its features, and an unsupervised clustering is performed which groups together the instances that have similar contextual features. Each group (or cluster) is treated as a distinct sense. A new word instance is assigned to its closest group based on its features. Note that, unlike the supervised case, we not have to rely on explicit sense labels from *WordNet*. Some authors make a terminological distinction to clarify this point, and use “word sense discrimination” as opposed to “word sense disambiguation”, while referring to unsupervised WSD.

A third and an interesting approach to WSD is to use a corpora of bilingual texts for disambiguation. Words that are polysemous in English are often not polysemous in Hindi, for example. The corresponding words in Hindi can then serve as sense identifiers. If we have a corpus of texts in English and their corresponding Hindi translations, we can use Machine Learning techniques to exploit contextual information to disambiguate polysemous English words.

16.2.5 Part of Speech Tagging

Determining the correct Part of Speech (such as noun, verb, adjective, adverb or determiner) of words in a sentence is a critical step for several *NLP* operations, including parsing. There are broadly two kinds of POS taggers: rule based taggers and stochastic taggers. Rule based taggers rely on a set of hand coded rules such as

IF preceding_word is DET, THEN current_word is NOT VERB

where DET stands for determiner (for example “the” or “a”).

While rule based taggers are efficient, they involve substantial knowledge acquisition overhead. To overcome this shortcoming, several supervised machine learning approaches have been explored to induce rules from annotated corpora. Stochastic taggers rely on the frequency of tags or sequence of tags, as estimated from a corpus.

The simplest scheme is based on the unigram model where the most frequent tag is assigned to a word. A bigram tagger recognizes that sequences such as “DET NN” are more likely than “DET VB”. Unlike rule based taggers that operate on a rigid set of rules, stochastic taggers aim at exploiting corpus statistics assigning the most likely sequence of tags to words in a sentence. A very popular scheme for stochastic tagging is the Hidden Markov Model (HMM) tagger. The idea behind HMM is discussed at length in Chapter 18. In the context of the current problem, the observed states are the words, and the POS tags constitute the hidden states. Stochastic taggers have been fairly successful with accuracies in the range of 95–96%.

16.2.6 Parsing

Traditional Parsing

Parsing is a well-studied area in the context of programming languages. However, as we noted earlier, parsers designed for programming languages are often not well suited for natural languages. This is because natural languages are inherently ambiguous, and at a syntax level, the number of valid parses for a sentence may sometimes exceed thousands. Knowledge of semantics, pragmatics or heuristics can be used to eliminate those that make no sense; this may still be a far cry, however, from realizing the goal of zeroing onto the one single parse that corresponds to the true intention of the author of the sentence. In traditional parsers, candidate parses are generated and a disambiguation module is used to choose the right parse based on additional information. There are some sentences that are fundamentally ambiguous, in the sense that even humans cannot disambiguate between candidate parses based on the available information.

Parsing a sentence (string) in a language needs knowledge of its grammar. There are systematic patterns in the sentence that emerge from the knowledge of grammar. For example, sentences typically have constituent phrases like noun phrases and verb phrases. Words in a constituent like a noun phrase ("the old man and the sea") occur close to each other.

Parsing can be viewed as a search problem where the search space is the set of trees consistent with a given grammar. There are two extreme ways of performing this search: top down and bottom up. These two terms are used in AI to refer to goal driven and data driven search respectively. In the task of packing bags for travel, we can start with the goal in mind and make a list of items that achieve that goal. Alternately, we can look around in the room and try to identify those items that we need to carry. The first approach is top-down, the second is bottom-up. In practice, a mix of both approaches is usually used.

```
S → NP VP
S → VP
NP → N
NP → PRON
NP → DET NOUN
NP → DET NOUN PP
NP → V
VP → VP NP
```

Figure 16.6 Rules of grammar.

In the context of parsing, a top-down parser is constrained by the grammar and a bottom-up parser by words in the sentence. Consider the

rules of grammar shown in Figure 16.6. Figure 16.7 illustrates the top-down process for the sentence “Fix the bugs”. All possible expansions of S, as suggested by these rules, are tried as shown in Level 1. The leaves of the trees in Level 1 define the new subgoals, which recursively lead to the generation of further levels. The generation stops when the leaves of the trees correspond to Parts of Speech. All trees whose leaves fail to match the words in the input sentence are rejected, the rest are recognized as syntactically valid parses of the sentence. In our example, the only parse tree that survives is shown at the bottom of Figure 16.7, in which “Fix” is a verb (phrase) and “the bugs” a noun phrase.

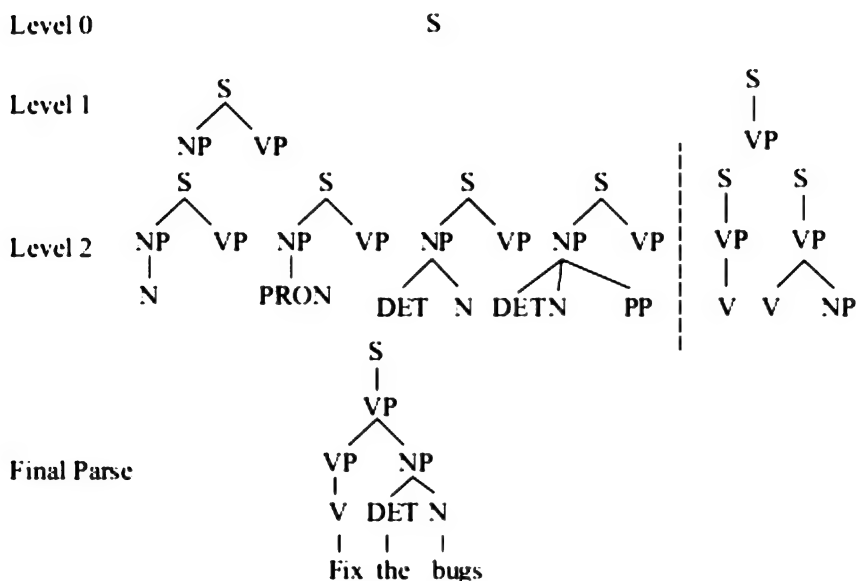


Figure 16.7 Top-down parsing.

Bottom up parsing starts by labelling each word in the input sentence with their POS tags. The number of such possible labellings (see Level 1 in Figure 16.8) is a function of the number of POS tags each word can potentially take. In the next step, the rules of the grammar are used to match the POS tags with the right hand side of the grammar rules. This results in the trees in Level 2. The process is continued with each level attempting to find rules whose right hand sides match the leaves of the trees in the earlier level. For a syntactically correct sentence, the process terminates when we have one or more trees rooted in S.

The bottom-up parser is inefficient, in that it generates trees that can never lead to an S. In contrast, the top-down parser sometimes produces trees that have no hope of matching the POS tags of the words in the sentence. Taking cues from the analogy of packing bags, it makes sense to exploit the best of both approaches. An idea called *bottom-up filtering* does just that. The basic idea is to preprocess the grammar rules and list

out the potential “left corner” POS tags for each nonterminal in the grammar. For example, given the grammar in Figure 16.6, for a string to be a Noun Phrase (NP), it needs to start with one of the following POS tags: Noun, DET or PRON. These define the left corner of NP. A top-down filter can exploit this information and filter out expansions that do not correspond with the POS tags of the input sentence. In the example in Figure 16.8, the expansions originating from $S \rightarrow NP VP$ in Level 1 clearly have no chance of leading to valid parses and thus are eliminated using bottom-up filtering.

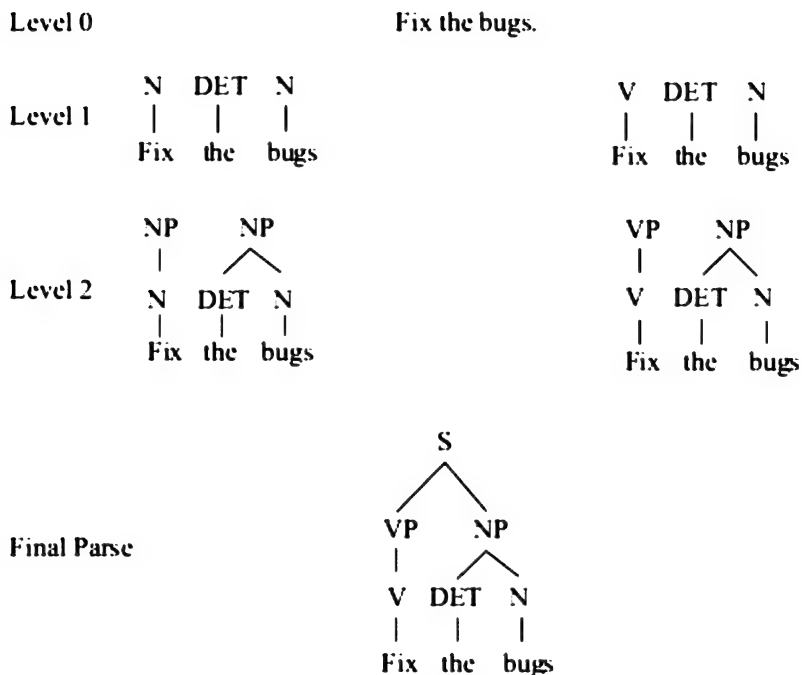


Figure 16.8 Bottom-up parsing.

Statistical Parsing

It is next to impossible to come up with a “perfect” grammar that accepts all English sentences that people find acceptable, and rejects those that people find unacceptable. Also, parsing a single sentence using a run-of-the-mill parser may lead to multiple parses, of which only one makes sense to people. These two observations lead us to speculate that it is perhaps more sensible to assign to a given parse of a sentence, a number between 0 and 1 (inclusive) that indicates the probability that the given parse is reckoned as a meaningful one by people. While Charniak did propose that syntactic analyses should be qualified by probabilities, the automata theorist Taylor L. Booth was the first to suggest that rules in

CFGs (context free grammars) should be assigned probabilities, giving rise to Probabilistic Context Free Grammars (PCFGs).

A simple example of PCFGs is shown in Figure 16.9 below. Note that the probabilities corresponding to the rules having the same nonterminal as antecedent add up to 1.

Let us use the grammar rules above to parse the following ambiguous sentence:

Deepa ate noodles with chopsticks.

$S \rightarrow NP VP (1.0)$	$NP \rightarrow NP PP (0.2)$	$V \rightarrow ate (1.0)$
$VP \rightarrow V NP (0.5)$	$NP \rightarrow Deepa (0.2)$	$PREP \rightarrow with (1.0)$
$VP \rightarrow VP PP (0.5)$	$NP \rightarrow noodles (0.3)$	
$PP \rightarrow PREP NP (1.0)$	$NP \rightarrow chopsticks (0.15)$	
	$NP \rightarrow spoons (0.15)$	

Figure 16.9 PCFG rules.

The two parse trees generated by the system are shown below. Using independence assumptions, the probability of a parse tree is simply estimated as the product of probabilities of all rules occurring in it. For the two parses Parse_1 and Parse_2 shown in Figure 16.10, the probabilities are estimated as shown below:

$$P(\text{Parse}_1) = P(S \rightarrow NP VP) \times P(VP \rightarrow V NP) \times P(NP \rightarrow NP PP) \times P(PP \rightarrow PREP NP) \\ \times P(NP \rightarrow Deepa) \times P(V \rightarrow ate) \times P(NP \rightarrow noodles) \\ \times P(PREP \rightarrow with) \times P(NP \rightarrow chopsticks)$$

$$= 0.0009$$

$$P(\text{Parse}_2) = P(S \rightarrow NP VP) \times P(VP \rightarrow VP PP) \times P(VP \rightarrow V NP) \times P(PP \rightarrow PREP NP) \\ \times P(NP \rightarrow Deepa) \times P(V \rightarrow ate) \times P(NP \rightarrow noodles) \\ \times P(PREP \rightarrow with) \times P(NP \rightarrow chopsticks)$$

$$= 0.00225$$

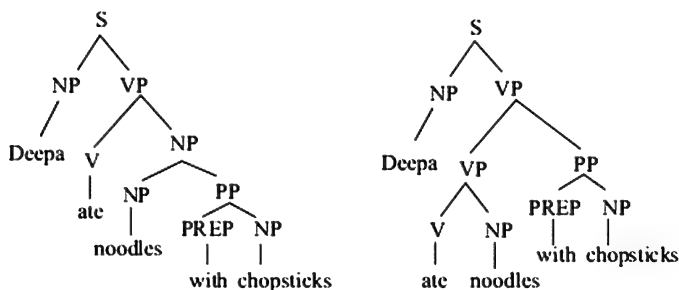


Figure 16.10 Two valid parses of the sentence 'Deepa ate noodles with chopsticks'.

The right parse tree (Parse_2) has a higher probability and would thus be preferred. The reader would have observed that the probabilities corresponding to each of the two parse trees are very low. This is expected, since the grammar can potentially generate an infinite number

of sentences which define the sample space, though the probabilities of very long sentences are expected to be close to zero. An example of such a long meaningless sentence accepted by the grammar above is “*Deepa ate noodles with chopsticks with spoons with chopsticks with spoons with Deepa*”.

An important question that we have not addressed so far is the following: How do we acquire the rule probabilities in a PCFG? The answer lies in using an annotated Treebank corpus (for example the Penn Treebank corpus) that contains a large number of sentences and their corresponding parse trees. The parse trees use rules of the form $a \rightarrow c$, where a is the antecedent (say NP) and c is the consequent (say DET N). The probability of the rule $a \rightarrow c$ is the ratio of the number of times the rule appears in the Treebank corpus and the number of times a occurs.

The approach mentioned above assumes a large corpus of sentences, each of which are associated with their correct parse trees. Construction of such treebanks involves substantial human intervention. It would be interesting to explore if we could start with a corpus that is only partially labelled with parse trees. This can be cast as a machine-learning problem where an inference has to be done in the presence of missing data. A classic approach is the Expectation Maximization (EM) algorithm outlined in Chapter 18. The application of EM algorithm to the problem of learning PCFG rules is discussed in (Manning et al., 1999).

16.2.7 Anaphora Resolution

In discourse, it is typical to point backwards to a previously mentioned entity. This phenomenon is referred to as anaphora, with the item referring backwards as an anaphor and the item being referred to as an antecedent. Let us consider a simple example:

Varun applied for several jobs in the banking sector.
Unfortunately, he failed to qualify in any of them.

Here, *he* in the second sentence is the anaphor pointing to the antecedent *Varun*. Detecting and resolving anaphora is important for NLP applications such as machine translation, text summarization and information extraction, which are covered in later sections of this chapter.

Anaphora resolution techniques typically identify a set of candidate antecedents, one of which is selected based on several cues. The first of such cues is gender and number. In the example above, the set of candidate antecedents for resolving the pronoun “he” include “Varun”, “several jobs” and “banking sector”. The resolution concludes that “he” must refer to “Varun”, as it cannot refer to “several jobs” because of a number conflict, nor can it refer to “banking sector” because of a gender conflict. A second cue is selectional restraint, wherein background knowledge about the candidate antecedents can help prune the

candidate set. An example is as follows:

Anuradha's friends baked cakes. They were delicious.

They, in the second sentence, must refer to *cakes* and not to the bakers; this is an example of semantic selectional restraint.

It may be noted that the term *anaphora* has a broader connotation than just pronoun resolution. Examples of other kinds of anaphora include lexical noun phrase anaphors (the IPL team Kings XI Punjab is referred to as just Kings XI), one anaphora ("*There are two teams. The one in red and yellow is East Bengal*") and zero anaphora ("*They played well and (they) won the match*"). For a more detailed account of various other kinds of anaphora and specific resolution approaches outside the ones mentioned above, refer to (Feldman et al., 2007).

16.3 Applications

16.3.1 Information Retrieval

Information Retrieval (*IR*) is the task of retrieving information from a given collection of documents that satisfies a certain information need. An obvious example is a search engine which is used by 85% of users when looking for some specific information. A central challenge in Information Retrieval is the uncertainty about the information need of the user, and also about the potential utility of the retrieved document(s) in meeting that information need. The system typically has to create underlying representations of the content of documents and match these up against the representation of the query. The system could fail because the representations of the query and of the documents fall short of modelling their actual information content, or because of the shortcomings of the matching process itself. A significant bulk of *IR* research has gone into formalisms/models for creating richer representations of the underlying semantic content or user intent.

A piece of text, at the surface level, is made up of words. At a deeper level, however, is its meaning and function. The lack of correspondence between deep and surface-level representations is a stumbling block for *IR*. Also, *IR* systems typically operate on large volumes of text, and this presents challenges in terms of devising algorithms that scale up on efficiency across time and space. In addition, ideas from research in Human Computer Interaction can be used to design interfaces that effectively elicit information needs of users. An example of this is the idea of relevance feedback which works as follows. The user is presented with a set of retrieved results and she offers her feedback by identifying those results which are relevant to the query, and those that are not. This feedback is used by the system to automatically reformulate the query and retrieve results again. A simple approach of query reformulation is

one that changes the query by including words from relevant documents and excluding words from irrelevant ones.

The quality of retrieval of an *IR* system is measured by *precision* and *recall*. Precision refers to the proportion of retrieved documents that are relevant. Recall refers to the proportion of relevant documents in the collection that have been retrieved. In the Venn diagram shown in Figure 16.11, the sets *RET* and *REL* refer to the set of retrieved results and the set of relevant results given a query, respectively. Precision and Recall can be defined in terms of these as follows:

$$\text{Precision} = \frac{|RET \cap REL|}{|RET|}$$

$$\text{Recall} = \frac{|RET \cap REL|}{|REL|}$$

As an example, consider an *IR* system that operates over a collection of 10,000 documents and retrieves 10 documents given a query *Q*. Let us assume that 6 of the retrieved documents are found to be relevant to *Q*. The total number of documents in the collection that are relevant to *Q* is 9. The precision is 6/10 and recall is 6/9. Note that evaluating recall is much harder than evaluating precision, since it is difficult to know a priori all documents that are relevant to a given query. The application needs determine whether an *IR* system should be tuned to maximize precision or recall. For example, precision may be more important in the context of a directory search system designed to fetch a telephone number, given a name or address. In contrast, recall is a better measure when it comes to evaluate yellow page search. An example is a system designed to retrieve all vegetarian restaurants around a specific location in a city. A typical user might be interested in going through most of the choices presented to her.

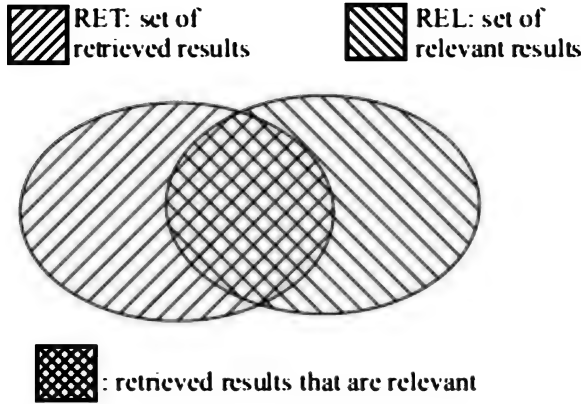


Figure 16.11 Illustrating the concepts of precision and recall.

It is commonplace in *IR* to view a document as a *bag* of words. The words are treated as independent of each other, and the order in which the words occur is ignored completely. While this clearly falls short of capturing deep meaning and the function of texts, this shortcoming is compensated by the gains in terms of retrieval and storage, especially in face of Web scale retrieval. Various *IR* models build on top on the bag-of-words model. Below, we illustrate how one such formalism, viz. *Vector Space Model*, is used to build a toy *IR* system. The first step is to conceptualize an n -dimensional space, where each of the dimensions corresponds to a word. Each document is represented as a vector in this a space. As an example, let us consider a document collection with just three documents D_1 , D_2 and D_3 . D_1 has words {cricket, Tendulkar, century}, D_2 has words {cricket, Dhoni, captain} and D_3 has words {cricket, Warne, bowler}. We construct a toy vector space with the 7 distinct words {cricket, Tendulkar, century, Dhoni, captain, Warne, bowler} as 7 dimensions. Each document is then represented as a vector in this space, where a 1 and 0 indicates the presence and absence of a word in a document respectively:

$$\vec{D}_1 : \{1, 1, 1, 0, 0, 0, 0\}$$

$$\vec{D}_2 : \{1, 0, 0, 1, 1, 0, 0\}$$

$$\vec{D}_3 : \{1, 0, 0, 0, 0, 1, 1\}$$

The retrieval process is inspired by the observation that documents sharing a lot of words map onto vectors that are close to each other in the vector space. If two documents share no words at all, the corresponding vectors are orthogonal. The similarity of two vectors is estimated by the cosine of the angle between vectors. For example, the cosine of the

angle between the vectors corresponding to \vec{D}_1 and \vec{D}_2 is

$$\text{sim}(D_1, D_2) = \frac{\vec{D}_1 \times \vec{D}_2}{|\vec{D}_1| \times |\vec{D}_2|}$$

Note that the denominator is the product of the norms of the two vectors. This has the positive effect of accounting for the dissimilarities in the lengths of the documents being compared. A typical Web query, for instance, may just be a few words long, while the documents it is compared against may have thousands of words.

In the example above, only the presence and absence of a term is considered in modelling the relevance of a term to a document. In practice, two other pieces of information can be used to arrive at better estimates of relevance. The first is a local measure of relevance. An example is the *term frequency*, defined as the number of times a term occurs in a document. The second is a global measure, in the sense that it attempts to capture the discriminating power of a term by examining its presence across the collection. Words like “the”, “of” and “a”, for instance, may occur in most documents in the collection, and thus have very little value in discriminating one document from the rest. An example of a global measure is the *inverse document frequency (idf)*, which is defined as follows:

$$idf = \log \left(\frac{N}{n} \right)$$

where N is the total number of documents in the collection and n is the number of documents in which the word occurs. The logarithm is used for scaling, since for many words, n is much smaller than N . Note that in the three-document collection shown above, the inverse document frequency of the word “cricket” is 0, symbolizing the fact that it has very poor discriminating power between the documents. The product of term frequency and inverse document frequency, referred to as the *tf-idf score* is often used to capture the overall relevance of a term to a document.

The steps involved in processing a document are as follows:

1. Tokenization The document is broken down into tokens after removing any irrelevant markups or metadata. A conscious choice needs to be made about handling punctuation marks and special symbols.

2. Stopword Removal Certain words, sometimes referred to as function words, play no important role in retrieval. Examples are articles like “a” or “the”, and prepositions like “on” and “in”. Typically, a stopwords list is used to identify and filter out such words. Additionally, a domain specific

stopword list may also be used.

3. Stemming This is used to reduce each occurrence of a word into its canonical representation, so that different variants of the same word (say “storing”, “stored”) reduce to their root form (“store”). Ideally, FSTs realizing a two-level morphology, as described in Section 16.2.2, should be used for this. In practice, however, simpler algorithms like Porter’s stemmer are often used.

4. Term Weighting The relevance of each term to a document is calculated using the *tf-idf* score described above. This gives us a vector corresponding to every document.

The same steps are repeated for the query as well. The query vector is then compared against each document vector using the cosine similarity score, and the documents are ranked in descending order and presented to the user.

The retrieval scheme described above has a practical limitation, in that it involves comparing the query vector sequentially with each document vector. A solution to this problem is the use of inverted file indices where a mapping is created from a word to the documents it occurs in. The advantage with this data structure is that we can restrict search to documents that contain at least one of the query terms, and thus avoid comparisons with completely unrelated documents. Figure 16.12 shows an example of an inverted file.

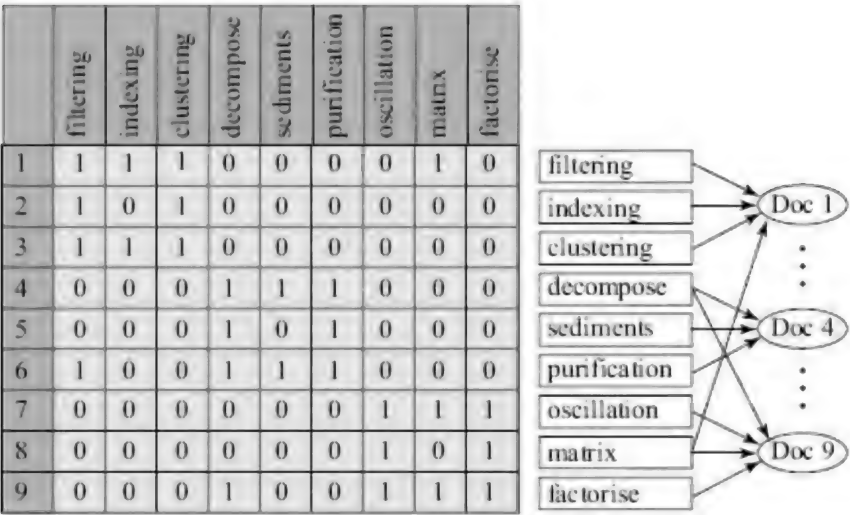


Figure 16.12 Inverted File Index for a toy document collection.

Outside the Vector Space Model, there exist a few other interesting *IR* formalisms like the Probabilistic Model and the Network Model, of which the former has attracted considerable attention in the *IR* community over

the last decade. However, this is out of the scope of the current chapter, interested readers are directed to (Rijsbergen, 1979) which has an introductory coverage of this topic.

16.3.2 Concept based Information Retrieval

The *IR* system, described in Section 16.3.1, has several limitations. The foremost among them is its assumption that words are unrelated to each other, which is obviously not true. We would expect a query on “heart attack” to retrieve documents on “cardiac arrest” as well. This motivates research on concept based *IR*, which aims at representing both documents and queries in terms of their underlying concepts, so that the retrieval is, by and large, robust to word choice variability. In recent years, there is a significant thrust on coming up with appropriate representations for concepts, and exploiting different knowledge sources for building these concept representations. At a broad level, there are three different knowledge sources: *linguistic*, *background* and *introspective*. Below, we discuss some prototypical concept mining approaches under these three categories.

Linguistic Knowledge

A lexical resource like *WordNet* can help in capturing relatedness between words. If “*cat licking mirror*” is presented as a query to an *IR* system, we would expect a document on “*animal biting glass*” to be marked as relevant. Several systems have been designed to realize this goal using *WordNet* which helps us with the knowledge that “cat” is a hyponym of “animal” and “mirror” is a kind of “glass”. One approach is to augment a query with all related terms from *WordNet*, and then do a traditional retrieval. A second approach is to use numeric measures of *WordNet* similarity, as explained in Section 16.2.3 above.

Background Knowledge

Humans often use a lot of background knowledge in answering questions posed to them. It is, for example, almost impossible to analyse an event pertaining to an Israel–Palestine conflict, unless someone has a good prior knowledge on the Middle East crisis. Lexicons like *WordNet* clearly fall short of capturing such knowledge. Interestingly, however, electronic encyclopaedias like *Wikipedia* have emerged as a rich storehouse of background knowledge. Each *Wikipedia* article, which can be treated as a distinct concept, can be represented by the words in its text. In addition, the articles are linked to each other and to Web pages outside *Wikipedia* using hyperlinks. Each article is categorised under a concept hierarchy that can also be exploited. (Gabrilovich et al., 2007) propose a technique

which they call *Explicit Semantic Analysis* (ESA), in which they treat each Wikipedia article as a concept. A vector space is then constructed with each concept as a dimension. Every word is represented in this space as a vector, which has a component 1 across a dimension, if the corresponding *Wikipedia* article contains that word, and has a component 0 otherwise. Since any given piece of text is simply a vector sum of its word vectors, it is easy to map both the query and the given set of documents to the concept space. Cosine similarities between the query and documents are computed and relevant documents are retrieved and ranked, as usual.

Introspective Knowledge

The idea here is to infer associations between words and phrases by investigating their co-occurrence patterns within a collection of documents. For example, the word “automobile” is likely to co-occur with words like “gear”, “chassis” or “suspension” in many documents, and can thus be inferred to be “semantically related” to these words. Statistical techniques can exploit these co-occurrence patterns to generate word clusters, which can in turn be used for query expansion. Co-occurrences have their limitations, however. (Lund et al., 1996) observe that near synonyms like *road* and *street* fail to co-occur in their huge corpus.

In a French corpus containing 24 million words from the daily newspaper *Le Monde* in 1999, (Lemaire et al., 2006) found 131 occurrences of *Internet*, 94 occurrences of *Web*, but no co-occurrences at all. This has motivated researchers to investigate ways of modelling *higher order* co-occurrence patterns between words. If words *A* and *B* co-occur in some documents and words *B* and *C* in some others, words *A* and *C* can be said to share a second order co-occurrence between them (via *B*). In the Section below, we briefly describe a Factor Analytic technique called *Latent Semantic Indexing* (LSI) that mines concepts from a document collection by exploiting higher order associations between terms.

Latent Semantic Indexing *LSI* was proposed as a technique for concept extraction in (Deerwester et al., 1990). The objective is to determine a set of underlying “factors” or concepts, that best explain the relationship between the terms and documents. This is not very different from the goal of most factor analysis research from the sixties to the nineties. What distinguishes *LSI* from most earlier approaches is its “two-mode factor analysis” which allows it to express both words and documents in terms of the same underlying concepts.

To start with, we have a term document matrix. Each element in that matrix is a weight showing the relevance of the term to the corresponding document. The first significant step in Linear Algebra is to view a matrix, such as this as an operator. This means that the matrix can act upon a

vector (when it is multiplied with that vector), and relocate it to a different position. For example, the square matrix³

$$M = \begin{pmatrix} 2 & -1 & 1 \\ -1 & 2 & -1 \\ 1 & -1 & 2 \end{pmatrix}$$

can act on the vector

$$\vec{A} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

and move it to a new location given by $M\vec{A}$:

$$M\vec{A} = \begin{pmatrix} 3 \\ 0 \\ 5 \end{pmatrix}$$

In the underlying geometry of the space, the action of a matrix M can be viewed as a combination of translation and rotation of \vec{A} in the general case. We are interested in characterizing a matrix M formally in terms of its properties that govern its action on vectors; the concept of *eigenvectors* does precisely that. We consider all vectors \vec{x} that, when acted on by M , stretch themselves to a different location $\lambda\vec{x}$, where λ is a scalar, but do not undergo any rotation. Thus,

$$M\vec{x} = \lambda\vec{x} \tag{16.1}$$

The vectors satisfying (16.1) are called eigenvectors, and each of these eigenvectors is associated with a corresponding value of λ referred to as an *eigenvalue*. We rewrite (16.1) as $(M - \lambda I)\vec{x} = 0$, where I is an identity matrix of dimensions matching M ; this is called the characteristic equation. Solving it in our example, we have the following three eigenvectors

$$\vec{v}_1 = \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}, \quad \vec{v}_2 = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \quad \text{and} \quad \vec{v}_3 = \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix}$$

associated with the eigenvalues $\lambda_1 = 1$, $\lambda_2 = 1$ and $\lambda_3 = 4$ respectively.

We now study the effect of M on any arbitrary vector \vec{x}

$$\vec{x} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

We can express \vec{x} as a linear combination of \vec{v}_1 , \vec{v}_2 and \vec{v}_3 .

The revised position $M\vec{x}$ is now given by

$$\begin{aligned} M\vec{x} &= M(1\vec{v}_1 + 2\vec{v}_2 + 3\vec{v}_3) \\ &= M\vec{v}_1 + 2M\vec{v}_2 + 3M\vec{v}_3 \\ &= \lambda_1\vec{v}_1 + 2\lambda_2\vec{v}_2 + 3\lambda_3\vec{v}_3 \end{aligned}$$

The interesting aspect of this rewrite is that we can see that the total effect of M on \vec{x} is expressed as a weighted combination of effects due to each eigenvector. Eigenvectors, having very small eigenvalues associated with them, have a small effect on the operation of M on \vec{x} . In the example above, the eigenvector associated with the eigenvalue $\lambda_3 = 4$ will have a more pronounced effect in characterizing M as an operator, compared to the two other eigenvectors, each associated with eigenvalue 1. This intuition is critical to our treatment of SVD below.

Moving on to a few more definitions, a family of a finite number of vectors is said to be *linearly independent* if none of them can be expressed as a linear combination of the remaining ones. The rank of a matrix M (not necessarily square) is the number of linearly independent columns (or rows) in it. It can be shown that the rank of a square matrix equals the number of its nonzero eigenvalues, counted with multiplicity.

We now look at an important result in factor analysis. For a given square matrix real valued $m \times m$ matrix M with linearly independent eigenvectors, we can obtain a factorization

$$M = U \Lambda U^{-1}$$

such that the columns of U are the eigenvectors of M , and Λ is a diagonal matrix whose diagonal elements are eigenvalues of M arranged in decreasing order. This result is due to Matrix Diagonalization Theorem (Strang, 2009) and applies to square matrices, but not to rectangular ones like the document-word matrix.

Previous attempts at factor analysis applied the idea to word-word matrices or document-document matrices, which are square. This is referred to as single-mode factor analysis. In contrast, two-mode factor analysis starts off with a rectangular word document matrix M of dimensions $m \times n$ (corresponding to m words and n documents), and rank r . The key apparatus is the Singular Value Decomposition (SVD) of M , which is given by:

$$M = U \Sigma V^T$$

where,

U is an $m \times m$ matrix whose columns are orthogonal eigenvectors of MM^T .

V is an $n \times n$ matrix whose columns are orthogonal eigenvectors of M^TM .

The eigenvalues $\lambda_1, \lambda_2, \dots, \lambda_r$ of MM^T are the same as eigenvalues of M^TM . The square root of these r eigenvalues, called singular values, are arranged in descending order along the diagonal of the matrix Σ , all other elements of which are set to 0.

We have seen before that small eigenvalues contribute less to the effect of the action of a matrix M on vectors. Extending this intuition to SVD, it is interesting to see the effect of considering only the top k singular values, and discarding the rest (flipping them to 0). Thus, the matrix Σ is shrunk to a $k \times k$ diagonal matrix Σ_k . We also delete the columns corresponding to low (and zero) singular values in U and V to obtain \hat{U} and \hat{V} respectively. \hat{U} , Σ_k and \hat{V} can now be combined to yield

$$\hat{M} = \hat{U} \Sigma_k \hat{V}^T \tag{16.2}$$

\hat{M} is a k -rank approximation to M . This result is pivotal to our discussion of LSI below.

Firstly, we note that SVD achieves dimensionality reduction. Let M be a document-word matrix, with each row representing a document. Geometrically, the rows of \hat{U} and \hat{V} are co-ordinates of points corresponding to documents and words mapped onto a k -dimensional

space. Typically, the axes are scaled using the k singular values to assign more importance to dimensions that are associated with high singular values. These reduced dimensional representations can then be compared against each other using the dot product or the cosine measure.

Secondly, it can be shown that \hat{M} is the best k -rank approximation to M in the least squares sense. The quality of an approximation M_A is measured by the Frobenius Norm of the “discrepancy” matrix $X = M - M_A$, which is given by

$$\|X\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n X_{ij}^2}.$$

The lower the value of $\|X\|_F$, the better the matrix M_A is, as an approximation to M . Viewing the low rank approximation problem as one of constraint optimization, it can be shown that, of all approximate matrices that satisfy the constraint that their rank is at most k , \hat{M} is the one that registers a minimum value for $\|X\|_F$. This conforms to our earlier intuition that removing very small singular values does not significantly affect M . The important thesis behind *LSI* is that the small singular values correspond to noise due to word choice variability (synonymy). \hat{M} is a less sparse representation compared to M that broadly retains the patterns of word association to documents, but at the same time “smoothes” it out to eliminate noise.

Thirdly, we note that the correspondence between low singular values and noise due to word choice variation is not accidental. Considering a square matrix M with two identical columns, we can eliminate one of these and still retain the same rank. This is a trivial case of feature selection. If instead, M had nearly identical columns, it would mean that the two corresponding features would have co-occurred similarly with documents. This would be true for closely related features like “Middle East” and “oil” which might appear in very similar contexts in a large document corpus. In such a case, it is intuitive that we can still go ahead with merging the two columns corresponding to the two features, and construct a new feature that averages or smoothes out the two original features. This is exactly what SVD achieves when it constructs a low rank approximation.

In this context, we make a critical distinction between the “true rank” and “effective rank” of a matrix. While the true rank takes into account all nonzero singular values, effective rank discards the very small ones. Thus, merging two closely related features changes the true rank but maintains the effective rank of the matrix. *The ability of SVD to identify*

“latent” co-occurrence patterns is the main reason for its improved effectiveness in retrieval tasks compared to the plain vector space model based on a bag of words. Also, the new features which are referred to as “concept” features are expected to be more robust indicators of meaning in comparison to the original feature set. This can be viewed as a step of feature extraction. It is important to note that extracted features can be expressed as a linear weighted combination of original features. There is another notable consequence of merging features: although LSI deals reasonably well with synonymy, (Deerwester et al., 1990) observes that the solution it offers to polysemy is at best partial. This is also confirmed by the results of their experiments. The problem lies in the fact that LSI forces a word to have a single representation in the concept space; thus a word with multiple meanings is represented as the weighted average of the different meanings. It is possible that none of the “real” meanings is close to the average, leading to a serious distortion.

Fourthly, both words and documents are treated in a uniform way by LSI. The concept features act as new dimensions, in terms of which both words and documents are represented. In Figure 16.13, which is an adapted version of Figure 18.3 from an online version of (Manning et al., 2008), we show an example of vectors spaces before and after LSI. Figure 16.13(b) shows how representations of words and documents obtained by LSI can be positioned in the new concept space. This allows us to visualize term and document clusters in the same space, and obtain interpretable descriptors of these clusters based on neighbouring words.

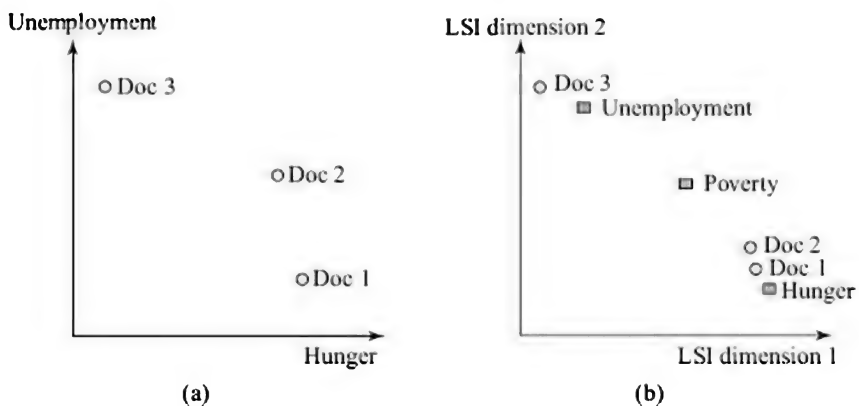


Figure 16.13 Vector spaces before and after LSI.

16.3.3 Information Extraction

The goal of *Information Extraction (IE)* is to automatically identify named entities like people, places and organizations, as well as events and relations between entities. Instead of attempting a full blown discourse

understanding, Information Extraction operates over a restricted domain, and makes use of specific domain knowledge to elicit only certain kinds of information from text.

The process of building an *IE* system begins with a knowledge engineer describing the domain of interest using a template. An example of a template is shown in Figure 16.14 below. A template is basically a set of attributes (slots) and corresponding values (fillers). An interesting aspect of the template idea in *IE* is its resemblance to structures proposed in literature on cognitive models of human memory. The process of understanding an article on an air crash involves invoking the template that captures salient aspects of a crash (when, where, number and type of casualties, for instance) and filling in this template, while also recording any significant additional information that may not have been prototypical of a crash (say, a miraculous escape). Once the filled-in template is recorded, it can be used to reconstruct a textual description of the crash.

<i>Flight No.</i>	<i>Air India 182</i>
Type of Carrier	Boeing 747-237B
Date	23/06/1985
Location of Crash	Irish Airspace
No. of the passengers and crew members on board	329
No. of survivors	0
Cause of accident	Bombing
Origin	Montreal
Stopover	London
Destination	New Delhi

Figure 16.14 An example of a filled-in template.

Once the template is defined for a specific domain, a sequence of steps is involved in extracting pieces of information from unstructured text and filling in the slots (Grishman, 1987). The first of these is *lexical analysis*, where the tokens are identified and labelled with features like Parts of Speech. This is followed by *Named Entity Recognition*, which involves identifying names of people and company names. Typically, this step makes use of simple rules ("Mr." precedes a person name and "Inc." is preceded by a company name) as well as a dictionary of company and people names. For accurate recognition, the effort involved in manually encoding the patterns could be substantial. This has motivated researchers to explore statistical classification approaches trained over a hand annotated corpus, as also Hidden Markov Models. Once the named entities are tagged, a *partial syntactic analysis* is carried out, wherein the constituents such as noun phrases and verb groups are identified. This is important because noun phrases (say "Air India Flight AI-673") often map onto the entities and verb groups (say "crash landed") are suggestive of

the events. In the fourth step, called the *scenario pattern matching*, the relations between entities and specific events are mined. This is done by matching the noun phrases and verb groups against a set of recorded patterns.

The central problem here is that the same event or relation can be expressed in a variety of ways in English. For example, “*the pilot made a safe landing*” and “*the plane was landed safely by the pilot*” should lead to the same patterns being triggered. As with capturing patterns using named entity recognition, while one option is to hand code these patterns explicitly, yet another is to make use of statistical techniques. While the steps discussed till now are concerned with sentence level processing, the final two steps involve looking across sentences. Consider the following two sentences: “*The pilot was experienced in flying under turbulent conditions. He made a safe descent.*” The “he” in the second sentence needs to be unified with “pilot”. This is achieved by *coreference analysis* which involves anaphora resolution as discussed in Section 16.2.7. The sixth and the final step is *inferencing and event merging*. Here, we use the information extracted to trigger inferencing, using a set of domain specific rules. An example of such a rule in Prolog syntax is shown below.

Failed_to_reach (X,Y) :-Destination (X,Y), Crashed (X).

This can be interpreted as: Flight *X* failed to reach destination *Y*, if it was slated for *Y* but crashed.

16.3.4 Machine Translation

Machine Translation (MT) refers to the process of automated translation of text from one language to another. Achieving human level translation quality is a holy grail in *NLP*, primarily because generating good translations needs a very good understanding of the source document. However, several interesting *MT* applications have been built and used commercially (Nilsson, 2010); often humans are involved in post-editing the machine generated output to improve its quality. In countries like India where a very small fraction of population can understand English, one of the particularly impressive applications of *MT* is in rendering the vast amount of material available on the Web to local languages (see also (Khemani, 2012)).

There are two broad schemes in *MT* systems. In *Direct MT*, a separate translator is built for each pair of source and target languages. In contrast, the *interlingua* based approach is founded on the idea of an intermediate language. Translators are built that convert text in a given language to the intermediate language, and vice versa. Given any source and target language pair, the source language text is first rendered into the intermediate language, which in turn is converted to the target. When compared to the direct approach, the interlingua based approach clearly

cuts down on the number of translators that need to be built, so that any of n given languages can be translated to any other. The concept of interlingua is also reminiscent of Chomsky's idea of a deep level structure shared by all languages.

There are several challenges in the way of developing successful *MT* systems. For one, we need to take into account differing word order in source and target languages. *NLP* techniques like Word Sense Disambiguation, anaphora resolution, and resolution of ambiguities also play important roles. Two important directions in the development of *MT* systems are *Rule Based MT* and *Corpus Based MT*. The latter has gained prominence in recent years, primarily because of the state-of-the-art performances obtained using significantly lower knowledge acquisition overheads.

Rule-based translation systems parse the text in source language to produce an internal representation which is then transferred to the target language and rendered into text. Sometimes, this threefold process is referred to as structural transfer. We show an example below, showing steps in transferring a sentence from English to Hindi.

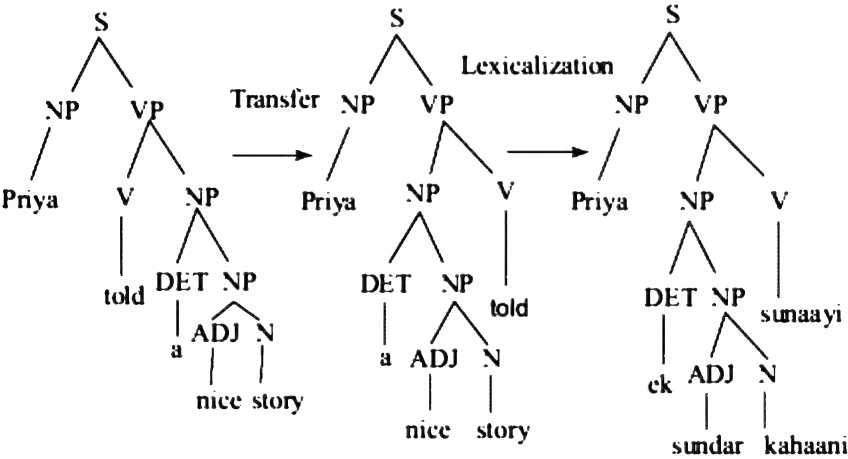


Figure 16.15 Steps in classical machine translation.

The input sentence is "*Priya told a nice story*". In Step 1, knowledge of English language grammar is used to create a parse tree of the sentence. In the structural transfer step, aspects of translation like word order reversals are taken care of. In the current example, a Subject-Verb-Object structure has been converted to a Subject-Object-Verb structure. The translated sentence is obtained in the final step of lexicalization and rendering, which typically makes use of target language grammar. A case marker "*ne*" may be added as part of post-processing to yield the output "*Priya ne ek sundar kahaani sunaayi*". Each of these steps rely on a large number of rules for handling issues like morphology, parsing in the face

of ambiguities, and for capturing knowledge in source-target language transfer.

The basic motivation behind statistical or corpus based *MT* is to use machine learning approaches to acquire the knowledge needed in translation. The source of this knowledge is a parallel corpus, which has a set of sentences in source and target languages. The idea of a noisy channel is relevant here. Given a Hindi sentence H , we intend to find an English sentence E that maximizes $P(E|H)$, which is, by Bayes' Rule, a function of two quantities $P(H|E)$ and $P(E)$. The former is called the translation model and the latter, the language model.

The idea of language models has received a lot of attention over the last decade. The basic idea is that given a large corpus, useful statistics may be obtained based on bigrams or trigrams, which are sequences of two or three words respectively. A language model in English may suggest that the probability of "to" following "going" (written as $P(\text{to}|\text{going})$) is higher than the probability of "on" following "going" ($P(\text{on}|\text{going})$). The translation model, on the other hand, consists of a set of parameters that define how words in the target language can be generated from the source language. Note that the translation model is only concerned about generating an appropriate set of target words and is agnostic to how they are arranged to give rise to the target sentence. This later aspect is taken care of, by the language model of the target language, which filters out any ungrammatical constructs. Adopting the convention of IBM Model 3⁴, the following are some of the parameters in the translation model:

- (a) Parameters like $t(\text{makaan} | \text{house})$, are translation probabilities, which gives the probability of producing "makaan" from "house."
- (b) Fertility parameters like $n(1 | \text{house})$, which gives the probability that "house" will produce exactly one Hindi word, whenever "house" appears.
- (c) Distortion parameters like $d(5 | 2)$ which gives the probability that an English word in position 2 (of an English sentence) will generate a Hindi word in position 5 (of a Hindi translation).

In practice, a richer distortion parameter like $d(5 | 2, 4, 6)$ is used in IBM Model 3, which is just like $d(5 | 2)$, except also given that the English sentence has four words and Hindi sentence has six words. Also, an additional set of parameters may be needed to capture the fact that a Hindi word may appear out of nowhere, i.e. when there is no corresponding English word.

We make two observations here. First, the number of parameters that need to be estimated is huge. Secondly, it appears quirky that such a strange scheme should ever work. Even the staunchest advocate of statistical *MT* would find it hard to justify that there is any remote resemblance to how humans do translation. It may also appear that a very large hand annotated parallel corpus of translated sentences would be needed to make robust estimates of these parameters. Even if such large corpora exist, they do not come with word-for-word alignments.

However, it is possible to obtain estimates from non-aligned sentence pairs using the *Expectation Maximization (EM)* algorithm, which is a technique for parameter optimization in the face of missing values. Details of *EM* algorithm are presented in Chapter 18. The basic idea is to start with arbitrary values of the parameters and keep refining them in successive iterations. Each pair of sentences in the source and target language, places constraint on the values the parameters can take. Thus, the parameter estimation problem reduces to an interesting problem, akin to the cracking of a Sudoku puzzle, given a set of standard constraints.

16.3.5 Text Summarization

An interesting application of *NLP* is in automatically generating summaries from natural language text. The generation could be *extractive* summarization, in which parts of the text (say, sentences) from the source are used verbatim to create summaries. In contrast, *abstractive* summarization is harder, in that it involves detailed interpretation of the text and regeneration of the substantive content.

The process of summarization can be broken down into three major steps. The first is *topic identification*, the goal of which is to return the highest scoring units (sentences) based on their suitability for inclusion in the summary. The suitability is estimated using a combination of factors, such as positional criteria (headings, titles and starting sentences may be more important than others), cue phrase indicator criteria (a sentence beginning with “the key contribution of this paper is” is likely to be important) and word frequency criteria (sentences having high frequencies of words that discriminate the text from others are likely to be more important). More such criteria, as well as an account of a comparison of their effectiveness based on empirical evaluations, are discussed in detail in Ed Hovy (Hovy, 2005). The second step is *interpretation* or *topic fusion*. This is an important step for abstractive summarization, and involves the use of information extraction approaches to fill in the slots of domain-specific templates, which capture the essential content that needs to be rendered into the summarised text. Thanks to knowledge acquisition bottlenecks, this is a hard problem and has been a major stumbling block in the way of building abstractive summarizers. The third and final step is *summary generation*, which uses *Natural Language Generation* techniques to render the filled-in templates resulting from the previous step to text. It may be noted that the role of interpretation and summary generation is minimal in the case of extractive summarization.

16.4 Natural Language Generation

NLG is complementary to *NLU*, in that it aims at constructing natural language text from a variety of nontextual representations like maps,

graphs, tables and temporal data. Such a conversion could be motivated by one or more distinct communication goals. An example of a communication goal is to make the information in a data like a map accessible to the blind. This can be achieved by having a speech synthesis system that renders the generated text into sound. A second goal could present the underlying data in a form that is more understandable to the lay user. This is true of systems that generate summaries of medical records, or explanations for reasoning performed by expert systems. Another potential application is automated tutoring systems. *NLG* can be used to automate routine tasks like generation of memos, letters or simulation reports. At the creative end of the spectrum, an ambitious goal of *NLG* would be to compose jokes, advertisements, stories and poetry.

An *NLG* solution may be appropriate in some situations, and less appropriate in others. For example, a graphical display like a bar chart may be more effective in depicting a comparative analysis of percapita income of Indian cities than a paragraph of verbose text. On the other hand, a textual description is more appropriate to present a technical argument explaining the differing per capita incomes. There are pragmatic considerations as well. A short piece of generated text may be easily sent over a mobile device, a picture may need more bandwidth. More often than not, we have to critically analyse the advantages of human authoring *vis a vis* automated generation. It could be that some of the knowledge required for generating text of acceptable quality is tacit and hence not available readily to the *NLG* system. In the other extreme, *NLG* may be an overkill in certain applications where a simple concatenation of strings (a canned text) may suffice. *NLG* is useful in situations where linguistic constraints need to be respected, and quality of generated text is important. Also, *NLG* systems have an advantage over humans, in that the texts generated by *NLG* systems are more consistent. Having said this, as with *MT* systems, many *NLG* systems may involve a final step of manual post-edit.

The traditional approach to *NLG* involves the following steps:

1. Document Planning (also referred to as Macroplanning) In this step, the *NLG* system identifies the content based on the communication goal and knowledge sources at its disposal. A text plan is then worked out, which organizes this content in a structured way. Often the plan is represented as a tree, with the leaves representing textual units like sentences. The nodes and edges of the tree are given interpretation according to the *Rhetorical Structure Theory* (RST) (Mann et al., 1987), which defines relations between units of text. For example, the relation *cause* connects the two sentences "*The hotel was costly.*" and "*We started looking for a cheaper option.*" Other such relations are *purpose*, *motivation* and *enablement*. The text is organized into two segments; the first is called a *nucleus*, which carries the most important information, and the second *satellites*, which provide a flesh around the nucleus. The way

a document plan is organized using rhetorical relations is illustrated in the example shown in Figure 16.16.

2. Microplanning The step aims at grouping the information into small units that can be mapped onto sentences. This, in turn, involves three main substeps. The first is *generation of referring expressions*. A decision needs to be made, for example, whether we should refer to Barack Obama as “the US president” or “Mr. Obama” or using the pronoun “he” instead. The second component of microplanning is *sentence aggregation*. Consider the sentences “*The hotel was located close to the conference venue. The staff was cordial. The hotel was expensive.*” Aggregation combines these sentences into one sentence: “*The hotel was located close to the conference venue and the staff was cordial, though it was expensive*”. Finally, microplanning involves *lexicalization*, which refers to the right choice of words to express a concept. While, for example, *rise* and *ascend* are near synonyms, it is unusual to describe the temperature as *ascending*.

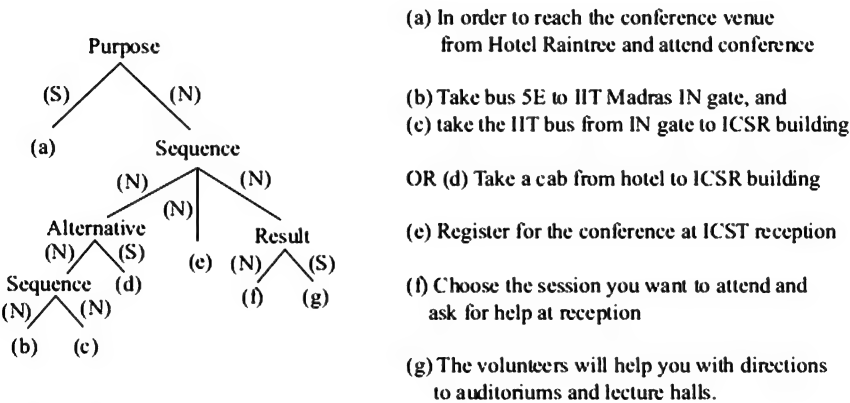


Figure 16.16 Rhetorical structures in Document Planning

3. Surface Realization The surface realization step deals with fixing the grammatical structure of the sentence and inserting the words in the appropriate slots of that structure. Decisions are made with respect to the insertion of prepositions or determining inflected or derived word forms.

4. Final Presentation This involves substeps like formatting, layout and punctuation.

It may be worthwhile to compare *NLU* against *NLG*. In *NLU*, the input is *NL* text, while *NLG* can accept varied forms of input. In *NLU*, the input is ambiguous, underspecified and ill-formed and the central concern is of managing the space of potential hypothesis about the underlying meaning. *NLG*, in contrast, deals with input that is relatively unambiguous, well specified and well-formed. The central concern in

NLG is choosing between different ways of expressing the underlying content, driven by communication goals and an understanding of the mental models of the target audience.



Exercises

1. Identify two ways in which *NLP* systems can make life easier for people with disabilities.
2. Give a *concrete* example to show that there are English sentences that cannot be recognized by regular grammar.
3. The ambiguity shown in the sentence of Figure 16.10 is said to result from Prepositional Phrase Attachment, since the prepositional phrase attaches to a noun phrase in one parse and to a verb phrase in another. Give another example of a sentence which has three valid parses due to differing attachment of a prepositional phrase. Show clearly the parse trees corresponding to each disambiguation.
4. Give an example to illustrate how Parts of Speech of context (neighbouring) words can be useful as features in Word Sense Disambiguation.
5. When your friend claims that LSI produces the best approximation to the original term document matrix, you would like to add two conditions to make her statement meaningful. What are these?
6. In the context of morphology, which of parsing and generation is harder? Give an example to justify your answer.
7. Name (a) one public domain tool that does morphological parsing, (b) one labelled corpus that can be used for supervised Word Sense Disambiguation, (c) one publicly available resource for estimating semantic relatedness between WordNet synsets, and (d) a treebank that is widely used in statistical Machine Translation.
8. Given four documents and pairwise distances between them in the vector space, their absolute positions in the vector space are known. True or false? Justify.
9. A search engine is designed to work over a collection of 1000 documents. In response to a query Q , the system retrieves 20 documents, of which 15 are found to be relevant. It is known from human judgements that the collection has 25 documents which are relevant to Q . Calculate the precision and recall of retrieval.
10. An Information Retrieval system has precision P and recall R with respect to a query Q . Think of a particle traversing a straight line segment of length unity, such that a distance α is covered with speed P and the remaining distance $(1 - \alpha)$ with speed R (such that $0 \leq \alpha \leq 1$). Show that the average speed of the particle over the entire stretch corresponds to F_α measure. In the light of this correspondence, can you suggest why the F_α measure is preferred over the arithmetic

mean of P and R ?

11. You are given a term document matrix of size 1000×200 . Can you apply the Matrix Diagonalization on this matrix to extract concepts? If yes, explain how, and show how this compares against mining concepts using SVD.
12. In stochastic Machine Translation from Hindi to English, we set out to find an English sentence that optimizes $P(E | H)$, but then use of Bayes' rule to estimate $P(H | E)$ and $P(E)$ instead. Why is estimating $P(H | E)$ any better than estimating $P(E | H)$ directly?
13. Can all the knowledge that a search engine needs about a movie be obtained from the pixels that make up the movie shots? Discuss.

¹ <http://www.departments.bucknell.edu/linguistics/synhead.html>

² <http://wn-similarity.sourceforge.net/>

³ Note that, unlike the example presented, all entries in a term document matrix are usually non-negative.

⁴ For a helpful online tutorial, refer to <http://www.isi.edu/natural-language/mt/wkbk.rtf>

Reasoning Under Uncertainty

Chapter 17

It is rarely that an agent has the privilege of reasoning with a complete and deterministic model of the world. This is especially true in a dynamic world which is changing all the time.

Uncertainty appears in different forms. The simplest form of uncertainty is the lack of complete information about what is *true* in the domain. This kind of uncertainty abounds in the real world. Generals have to fight battles without the benefit of knowledge of the strength and weaponry of the enemy. In older times, their battlefields were more like chessboards. Over the last hundred years or so, they have known less and less about the adversary, and more recently not even knowing who the enemy is. The reason why we hop from shop to shop, or portal to portal, is to find out who is selling what product at what price. Much of our everyday activity is oriented towards gathering information. Knowing facts is of considerable value, as it allows us to make decisions in a more rational manner. Our knowledge of the world around us is never complete though, and we are often compelled to arrive at conclusions in the face of incomplete and uncertain knowledge.

Another reason why we have to often reason with uncertain knowledge is that our knowledge of relations between categories is uncertain. This may be because of our penchant to generalize and express relations that are *generally* true, whereas they may not be *universally* true. For example, the general statement “birds can fly”. We will look at reasoning with such relations in the section on Default Reasoning. The other reason is that often relations between categories are directional in nature because there is a causal connection between them. For example, if one has malaria then one will also have fever. However, the need for reasoning often comes in asking about the converse relation. Given that one has fever, one may want to find out whether one has malaria or some other affliction. We will explore such connections in the section on Abductive Reasoning.

Often actions in the real world do not have deterministic effects. You may throw the basketball towards the basket but, unless you are Magic Johnson, there is a chance that it may not do the intended. We will look at planning with such stochastic actions later in the chapter.

Very often, the information we have about the world is not exact. This is especially true of numeric data. Consider a pan on the stove containing water. How does one describe the temperature of the water? The

simplest way to do so is to specify the temperature as a numerical quantity. That is what a scientist would do if she were doing an experiment. All of us, on the other hand, reason about hot water in more *qualitative* ways.

Using a predicate in logic, one can assert that the water is hot, or it is not. The semantics of classical logic dictates that we think of the water in the pan as an object, and place it either in the set of objects that are hot, or outside it. However, such clear cut demarcation into crisp sets is not always meaningful. How would one decide which set an object belongs to? Is it based on a threshold, say 70° Celsius? Then what about 69.9° Celsius?

Some techniques that have evolved to avoid crisp division of objects with continuously changing properties are the use of Fuzzy Sets and Rough Sets (see also Chapter 13). Fuzzy sets were devised to allow us to use linguistic terms like *hot*, *tall*, *heavy*, etc. to refer to such properties. The difference is in the semantics, which does not map to crisp sets. Rather, the “sets” themselves are defined with varying degrees of membership. Water at 95° Celsius belongs more to the set of hot objects, than water at 70° Celsius. Rough sets introduced a notion of modality into memberships. Thus, one can distinguish objects that definitely belong to a set from ones that may belong to it, and which definitely do not belong to it.

Qualitative reasoning allows us to define *quantity spaces* that are determined by the properties themselves, or some task at hand. For example, water between room temperature and boiling point can be thought of as being same. Beyond the boiling point, there is a qualitative change; it becomes a vapour. We look at qualitative reasoning a little later in this chapter.

We begin within the logical framework, looking at the task of making inferences in the face of incomplete information. Our main goal here is to explore ways in which we can exploit general knowledge to make inferences that will be true, more often than not.

17.1 Default Reasoning

One facet of reasoning under uncertainty is known as default reasoning. This involves making inferences that are *plausible* or *likely*, but not necessarily entailed by the knowledge base. The need for default reasoning arises because of our desire to generalize connections between categories, to express them in a succinct manner. Birds fly, leaves are green, clouds indicate rain, and computer science students are bright. The trouble with such generalizations is that they have exceptions. While it is generally true that birds fly, and that is indeed useful knowledge to have, it is not true that *all* birds fly. A universal statement like the one below is simply not adequate.

$$\forall x (\text{Bird}(x) \supset \text{Flies}(x))$$

The moment we come up with an exception, for example a bird called Peppy who cannot fly being a penguin,

$$\text{Bird(peppy)} \wedge \text{Penguin(peppy)} \wedge \neg \text{Flies(peppy)}$$

our knowledge base becomes unsatisfiable (see Exercise 12.20). And we are forced to throw away the universal implication that all birds fly.

Instead, we would like to have a mechanism by which, given a knowledge base, we can make the set of plausible inferences, with the caveat that if the knowledge base grows then some of the inferences may not hold. This implies that the set of inferences that we can make does not grow monotonically with what we know, and could in fact become smaller when we add more facts. This form of reasoning is called *nonmonotonic reasoning*, because the set of inferred sentences does not grow monotonically with the set of known facts. In the above example, if the only facts we had about Peppy were that it was a bird then it would be reasonable to infer that it could fly. Later, if we come to know that Peppy was a penguin, and we believed that penguins do not fly, we would no longer be justified in inferring that Peppy can fly, and would have to withdraw the belief.

We look at some of the approaches that have been proposed by researchers, though none of them stands out (Brachman and Levesque, 2004). The approaches vary in the kind of assumptions they make and the manner in which they use them.

17.1.1 Closed World Assumption

The simplest approach to dealing with incomplete knowledge is to make the assumption that the agent knows everything about the world, and anything that is not known or cannot be inferred from what is known is false. This is a kind of *negation by failure*, in which all known individuals in the domain, corresponding to constants in the language, only belong to categories they are known to belong to. In other words, this assumption *minimizes* the extensions or extent¹ of all primitive categories.

Let us assume a domain of youngsters and some information about who are friends, with the knowledge that friends (usually) chat with each other. We assume the *Friend* relation is symmetric. Consider the following first order logic knowledge base,

```
KB1:  { $\forall x \forall y$  (Friend(x,y)  $\supset$  Friend(y,x))
         $\forall x \forall y$  (Friend(x,y)  $\supset$  Chat(x,y))
        Friend(aditi, shubhgata), Friend(aditi, jennifer),
        Friend(jennifer, vinayak), Friend(shubagata, subun)
        }
```

Then the procedure under the closed world assumption augments the knowledge base KB_1 to create a new knowledge base KB_1^+ by adding negative atomic sentences where it can.

```

KB1+ : { (∀x ∀y (Friend(x,y) ⊃ Friend(y,x))
          ∀x ∀y (Friend(x,y) ⊃ Chat(x,y))
          Friend(aditi, shubhgata), Friend(aditi, jennifer), Friend
                                          (jennifer, vinayak),
          Friend(shubhgata, subun),

          ¬Friend(aditi, vinayak), ¬Friend(aditi, subun),
                                          ¬Friend(vinayak, subun)
          ¬Friend(subun, vinayak), ¬Friend(subun, aditi),
                                          ¬Friend(subun, jennifer),
          ¬Friend(shubhgata, vinayak), ¬Friend(shubhgata, jennifer),
          ¬Friend(jennifer, subun), ¬Friend(jennifer, shubhgata),
          ¬Friend(vinayak, aditi), ¬Friend(vinayak, shubhgata),

          ¬Chat(aditi, vinayak), ¬Chat(aditi, subun), ¬Chat(vinayak, subun)
          ¬Chat(subun, vinayak), ¬Chat(subun, aditi), ¬Chat(subun, jennifer),
          ¬Chat(shubhgata, vinayak), ¬Chat(shubhgata, jennifer),
          ¬Chat(jennifer, subun), ¬Chat(jennifer, shubhgata)
          ¬Chat(vinayak, aditi), ¬Chat(vinayak, shubhgata)
        }

```

The set of sentences one is entitled to believe in under the *Closed World Assumption* (CWA) are the sentences that are entailed by KB_1^+ . In fact, we can define a new notion of entailment under CWA as follows,

$$KB \models_{CWA} \alpha \quad \text{iff} \quad KB^- \models \alpha$$

where KB^+ is defined as,

$$KB^+ = KB \cup \{ \neg P \mid P \text{ is a ground atomic formula and } KB \not\models P \}$$

Reasoning under the closed world assumption is thus based on the premise that all ground atomic formulas that are true have been asserted in the knowledge base, or are entailed in it. Recall that the model for a set of sentences in a language is a domain and an interpretation such that all the sentences are true in that domain under the interpretation (see Chapter 12). The interpretation of a predicate is a relation of appropriate arity in the domain. From this perspective, one can see that the closed world assumption aims to find minimal models, in which the extent of all atomic predicates is as small as can be given the knowledge base. In the above example, this means that only those pairs who have been asserted as friends are in fact friends. One could find models in which other sentences are true, for example in some model, Subun and Vinayak could be friends. But CWA ignores that model and selects a model in which the extent of the *Friend* relation is minimum.

However, as is often the case in computing with the disjunction operator, there can be some difficulties. Consider the following knowledge base.

$$KB_2 = \{ \text{Friend(aditi, shubhgata)} \vee \text{Friend(aditi, jennifer)} \}$$

Now,

$$KB_2^+ = \{ \text{Friend(aditi, shubhgata)} \vee \text{Friend(aditi, jennifer)}, \\ \neg \text{Friend(aditi, shubhgata)}, \neg \text{Friend(aditi, jennifer)} \}$$

However, the extended knowledge base is inconsistent. This is because the procedure to add negative atomic sentences adds both the sentences, since it is unable to infer that either is true. One can get around this problem by making a *Generalized Closed World Assumption* (GCWA) in which an atomic formula that participates in a disjunction can be negated, only if the other literals in the disjunction are entailed by the knowledge base. The extended knowledge base KB° under GCWA is defined as,

$$KB^\circ = KB \cup \{\neg P \mid P \text{ is a ground atomic formula and,} \\ \text{if } KB \models (P \vee Q_1 \vee Q_2 \dots \vee Q_n) \\ \text{then } KB \models (Q_1 \vee Q_2 \dots \vee Q_n)\}$$

Given the new procedure we have,

$$KB_2^\circ = \{(Friend(aditi, shubhgata) \vee Friend(aditi, jennifer))\}$$

and one is (correctly) unable to make a prediction about individual friendships. But if we had the fact $(Friend(aditi, shubhgata))$ in KB_2 as well, then the extension would be different. That is given,

$$KB'_2 = \{(Friend(aditi, shubhgata) \vee Friend(aditi, jennifer)), \\ Friend(aditi, shubhgata)\}$$

we would have,

$$KB'^\circ_2 = \{(Friend(aditi, shubhgata) \vee Friend(aditi, jennifer)), \\ Friend(aditi, shubhgata), Friend(aditi, jennifer)\}$$

There is one small point one needs to make about reasoning with the closed world assumption. Suppose the knowledge base KB_1 has a student Divya for whom there is no known friend in the class. How should a system answer the query below?

$$\exists x Chat(x, divya) ?$$

Should the system answer *yes* or *no*? On the face of it, the answer should be *no* because that is what the knowledge base suggests. However, to answer with a definitive *no*, the system will have to make a *domain closure* assumption. The domain closure assumption says that the only people in the domain are those that have been explicitly mentioned. For the KB_1 , this assumption is expressed as,

$$\forall x (x = aditi \vee x = divya \vee x = jennifer \vee x = shubhgata \vee x = subun \vee x = vinayak)$$

Given this assumption along with the closed world assumption, the knowledge base KB_1 and the new statement about Divya, the answer to the query is *no*.

Domain closure can have problems of its own. Suppose we further

add the following statement to the knowledge base.

$$\exists x (\text{NewStudent}(x) \wedge x \neq \text{divya})$$

Then this unnamed student does not make it into the domain closure statement. Yet as we can see, our conclusion is not longer justified. For all we know the new student might be a friend of Divya. To infer now that Divya has no friend, we would also have a closure statement saying that we have said all that can be said about the *Friend* relation. This is a little easier in the second form of default reasoning we shall see in the next section.

17.1.2 Circumscription

Circumscription, devised by John McCarthy, is an approach that aims to minimize the extent of only some predicates (McCarthy, 1980; 1986), (Lifschitz, 1985; 1994). Traditionally, these predicates characterize *abnormality* with respect to the intended default inference, but circumscription itself can be done over any set of specified predicates.

As observed earlier, a universally quantified relation between birds and flying ability can run into problems making the knowledge base inconsistent when exceptions occur. The solution to this problem, as proposed by McCarthy, adds another clause to the antecedent saying that in addition to being birds, the individual should not be abnormal. This clause is intended to catch the abnormal cases.

$$\forall x (\text{Bird}(x) \wedge \neg \text{Ab}(x) \supset \text{Flies}(x))$$

In this formulation, the abnormal birds are in fact those that cannot fly. Default reasoning with circumscription aims to minimize the extent of the abnormality predicates. That is, one assumes that individuals are by and large normal (not abnormal), with respect to the intended conclusion. The statement can be read as "If something is a bird then it can fly, unless it happens to be abnormal with respect to the ability to fly". And given that we would like to normally associate birds with flight, we assume that the set of abnormal individuals is as small as can be.

Let us look at the example of people chatting again. Now the author has a friend who does not chat with him, so we know that the universal does not hold. Let us add the fact that friends chat as long as they not "isolated"².

```
KB3: {       $\forall x \forall y (\text{Friend}(x,y) \supset \text{Friend}(y,x))$ 
 $\forall x \forall y (\text{Friend}(x,y) \wedge \neg \text{Isolated}(x) \wedge \neg \text{Isolated}(y) \supset \text{Chat}(x,y))$ 
  Friend(aditi, shubhgata), Friend(aditi, jennifer),
  Friend(jennifer, vinayak), Friend(shubagata, subun),
  Isolated(jennifer)
}
```

Unlike closed world reasoning, we do not add any statements to the

knowledge base, but we can still characterize the set of sentences that follow from the assumption that the interpretation or extent of the *Isolated* predicate is minimized.

However, we are now beyond the realm of first order logic, since we need to talk about the extensions of predicates to be minimal. One way to express this process of minimization is to resort to second order logic and say that the predicate in question cannot be replaced with an equivalent predicate with smaller extent, without falsifying the given knowledge base. Let us say that the predicate we want to circumscribe is P . The circumscription of a formula (or knowledge base) Φ with respect to the predicate P written as $\text{Circ}[\Phi; P]$ can be defined as follows³ (Lifschitz, 1994),

$$\text{Circ}[\Phi; P] \equiv \Phi \wedge \neg \exists Q (\Phi(Q) \wedge (Q < P))$$

Here, $\Phi(Q)$ is the formula Φ with all occurrences of P replaced by Q . The ordering $<$ between predicates is defined as follows:

1. $(Q \leq P)$ if $\forall x (Q(x) \supset P(x))$
2. $(Q = P)$ if $\forall x (Q(x) \equiv P(x))$
3. $(Q < P)$ if $(Q \leq P)$ and $(Q \neq P)$

This says that the interpretation or extension of Q is strictly contained in the extension of P . The above definition can be extended when predicates P and Q have arity greater than one.

The definition can be read as follows. The circumscription of Φ with respect to the predicate P is Φ when the extension of the predicate P is as small as possible. This is ensured by the fact that P is chosen such that there is no “smaller” predicate Q which makes $\Phi(Q)$ true.

In our example, P and Q are variations of the *Isolated* predicate, the one we seek to minimize the extent of. They differ on which individuals they are true for. Let us say as an example that $I(P_{\text{Isolated}}) = \{\text{Jennifer, Subun}\}$. Now there exists a Q_{Isolated} which is true only for $x = \text{Jennifer}$, and satisfies the ordering condition 3 in the definition of circumscription. That is, $Q_{\text{Isolated}} < P_{\text{Isolated}}$. Further, the KB_3 is true for Q_{Isolated} . Thus, this version of P_{Isolated} does not participate in the circumscribed knowledge base.

One can observe that in the circumscribed knowledge base, the formula P_{Isolated} has to be true *only* for $x = \text{Jennifer}$. Then one cannot find a “smaller” predicate such the knowledge base is true. And this circumscribed knowledge base entails $\text{Chat}(\text{aditi, shubhagata})$. This is still contingent on the fact that Shubhagata and Jennifer do not refer to the same individual, and this can be ensured by make a Unique Name Assumption (UNA) which says that every constant is a unique name for some individual.

An alternative way of specifying the predicate with the smallest extension is to consider interpretations and choose the “smallest”

interpretation defined as follows (Brachman and Levesque, 2004).

Let $\mathfrak{I}_1(D, I_1)$ and $\mathfrak{I}_2(D, I_2)$ be two interpretations that agree on all constants and functions of the language. We define the relation \leq as follows,

$$\mathfrak{I}_1 \leq \mathfrak{I}_2 \text{ iff for every predicate } P \text{ being circumscribed } I_1(P) \subseteq I_2(P)$$

And, $\mathfrak{I}_1 < \mathfrak{I}_2$ iff $\mathfrak{I}_1 \leq \mathfrak{I}_2$ and $\mathfrak{I}_2 \not\leq \mathfrak{I}_1$.

We can now define entailment \models_{\leq} under circumscription as⁴,

$$KB \models_{\leq} \alpha \text{ iff for every interpretation } \mathfrak{I} \text{ such that } \mathfrak{I} \models KB \text{ either}$$

$$KB \models \alpha \text{ or there is an interpretation } \mathfrak{I}' \text{ such that } \mathfrak{I}' < \mathfrak{I} \text{ and } \mathfrak{I}' \models KB.$$

If the predicate being circumscribed is *Isolated*, then we can also say equivalently that,

$$KB \models_{\leq} \alpha \text{ iff } \text{Circ}[KB; \text{Isolated}] \models \alpha$$

Let us look at the KB_3 and pose the query $\text{Chat}(\text{aditi}, \text{shubhagata})$? There are four interpretations that may be relevant here and we have to look at *every* one of them.

$$\mathfrak{I}_1: \text{aditi} \notin I_1(\text{Isolated}) \text{ and } \text{shubhagata} \notin I_1(\text{Isolated}).$$

Here, the case is that,

$$\text{Friend}(\text{aditi}, \text{shubhagata}) \wedge \neg \text{Isolated}(\text{aditi}) \wedge \neg \text{Isolated}(\text{shubhagata})$$

and $\forall x \forall y (\text{Friend}(x, y) \wedge \neg \text{Isolated}(x) \wedge \neg \text{Isolated}(y) \supset \text{Chat}(x, y))$

Therefore, $\text{Chat}(\text{aditi}, \text{shubhagata})$

$\mathfrak{I}_2: \text{aditi} \in I_2(\text{Isolated})$ and $\text{shubhagata} \notin I_2(\text{Isolated})$.

Here, the above deduction does not apply but it is the case there exists a smaller interpretation \mathfrak{I}_2' in which $\text{aditi} \in I_2(\text{Isolated})$ and

$$\mathfrak{I}_2' \models KB_3$$

$\mathfrak{I}_3: \text{aditi} \notin I_3(\text{Isolated})$ and $\text{shubhagata} \in I_3(\text{Isolated})$.

Here again, the deduction from case 1 does not apply but it is the case there is a smaller interpretation \mathfrak{I}_3' in which $\text{shubhagata} \in I_3(\text{Isolated})$

and $\mathfrak{I}_3' \models KB_3$

$\mathfrak{I}_4: \text{aditi} \in I_4(\text{Isolated})$ and $\text{shubhagata} \in I_4(\text{Isolated})$.

Here again, the deduction from case 1 does not apply but it is the case there is a smaller interpretation \mathfrak{I}_4' in which $\text{aditi} \in I_2(\text{Isolated})$,

$\text{shubhagata} \notin I_4(\text{Isolated})$ and $\mathfrak{I}_4' \models KB_3$

Observe that \mathfrak{I}_2' , \mathfrak{I}_3' , and \mathfrak{I}_4' are all equivalent to \mathfrak{I}_1 , as far as membership of Aditi and Shubagata in the extension of *Isolated* is

concerned. Here we have covered all four relevant cases, and ignored the interpretations in which Subun does or does not belong to the interpretation of *Isolated*.

The key point is that for *every* interpretation either the goal formula must be *true* or there must be a smaller interpretation in which the knowledge base is *true*. Consider the query *Chat(aditi,jennifer)?* This query fails because there is an interpretation in which only Jennifer belongs to the image of *Isolated*. In this interpretation, the formula *Chat(aditi,Jennifer)* is not entailed. Further, we cannot find a smaller interpretation in which the KB_3 is true. If we remove Jennifer from the set of isolated people, the formula *Isolated(Jennifer)* in KB_3 becomes *false* and KB_3 is not *true*.

To summarize, $\text{Circ}[\Phi; P]$ is defined as follows. Consider all possible interpretations of the predicate P . These interpretations organize themselves into a lattice structure defined by the ordering (or subset) relation defined above. Extract from this lattice a sublattice in which the given knowledge base Φ . The set of minimal elements of this sub-lattice is the circumscription. If a given formula α is entailed in all these minimal

interpretations, then we accept that $\Phi \models_{\leq} \alpha$.

The concept of *Circumscription* is not confined just to “abnormal” predicates. It is only a mechanism to specify which predicates need to be minimized in the extent. The reader will recall that in our study of the *Event Calculus* in Chapter 13, we had employed *Circumscription* to make the assumption that only the specified events had happened, and that events had only the specified effects. In our example KB_1 , with addition of Divya and the new unnamed student, we could have used *Circumscription* of the augmented KB_1 with respect to the *Friends* predicate to be able to answer that there was no one Divya chatted with.

Circumscription has its own problems. Imagine that in our domain of people who make friends and chat with them on the Internet, there is a category⁵ of people called *IDoNotChat* who, as the name suggests, do not chat. This makes them “isolated”. Let us assume that *there are* such people, though the knowledge base does not say so explicitly. And let us assume, for the sake of illustration, that Jennifer is no longer in *Isolated*. The resulting knowledge base KB_4 looks like,

```
KB4: {   $\forall x \forall y (Friend(x,y) \supset Friend(y,x))$ 
         $\forall x \forall y (Friend(x,y) \wedge \neg Isolated(x) \wedge \neg Isolated(y) \supset Chat(x,y))$ 
         $\forall x (IDoNotChat(x) \supset Isolated(x))$ 
        Friend(aditi, shubhgata), Friend(aditi, jennifer),
        Friend(jennifer, vinayak), Friend(shubagata, subun),
        }
```

Now, given the same query *Chat(aditi, shubhgata)* $\text{Circ}[KB_4; Isolated]$ constructs a model in which there are no *Isolated* people.

$$KB_4 \models_{\leq} \neg \exists x \text{ Isolated}(x)$$

But this means that there can be no members in the *IDoNotChat* category. The problem is that in the process of minimizing the set *Isolated* (so that both Aditi and Shubhagata do not fall in the set), the proverbial baby has been thrown out with the bath water and the legitimate set of *IDoNotChat* people have been ignored.

There has been no convincing solution to this problem. One suggestion has been that some categories be kept fixed in size. In this example, we could say that the set of people in the category *IDoNotChat* is immutable. This would of course mean that the smallest set of *Isolated* people *must* contain the people in the *IDoNotChat* set.

However, once we admit that there are some people in the set *Isolated* we are back to square one and we cannot convincingly answer “yes” to the *Chat(aditi, shubhagata)* query. This is because we *do not know* that one (or both) of them does not belong to the category *IDoNotChat*. And we have bound ourselves to not minimize this set. This problem becomes more acute if we admit in our database the fact $\exists x(\text{IDoNotChat}(x))$ saying that this category is not empty. We can only conclude that they do chat, but only if they do not belong to this (immutable) set of *IDoNotChat* people.

Circumscription puts the task of default reasoning in a logical reasoning mould. Once the predicates to be circumscribed are identified, and the minimal extensions chosen, the rest is reasoning in first order logic. In the next section, we look at an approach which is known with the title of logic, but in fact is less “logical” in nature.

17.1.3 Default Logic

The closed world assumption adds the negation of all ground atomic formulas that are not entailed by the knowledge base. The idea in *Default Logic* is to extend the knowledge base with an appropriate set of (positive) ground atomic sentences (Reiter, 1980). The formulas that can be added are determined by *default rules*. The set of default rules \mathcal{D} , together with the first order theory \mathcal{F} , form a *default theory* $\langle \mathcal{F}, \mathcal{D} \rangle$ that augments the first order theory \mathcal{F} .

The default rules, as described by Raymond Reiter, have the following format,

$$\langle \alpha: \beta / \delta \rangle$$

The rule can be read as follows.

If α is *true* and it is consistent to believe β then δ may be added as an assumption (or inference). Here, α is the *prerequisite* for believing δ and β is a *justification*. The justification says that it is consistent to believe β , which means that there is no support for $\neg\beta$.

The association between being friends and chatting from the previous section can be captured in the following default rule.

$$\langle \text{Friend}(x, y) : \neg\text{Isolated}(x) \wedge \neg\text{Isolated}(y) / \text{Chat}(x, y) \rangle$$

The variables in the rule are treated as free variables and the rule can be viewed as a collection of ground rules of the form,

$$\langle \text{Friend}(\text{aditi}, \text{shubhagata}) : \neg\text{Isolated}(\text{aditi}) \wedge \neg\text{Isolated}(\text{shubhagata}) / \text{Chat}(\text{aditi}, \text{shubhagata}) \rangle$$

The rule says that *if* it is consistent to believe that Aditi and Shubhagata are not isolated then one can assume that they chat, *given* that they are friends. And since KB_3 does not entail either $\text{Isolated}(\text{aditi})$ or $\text{Isolated}(\text{shubhagata})$, it is consistent to believe the opposite, and add $\text{Chat}(\text{aditi}, \text{shubhagata})$ to the set of beliefs.

Adding a new formula *extends* the default theory and an *extension* of a default theory is the maximal set of (consistent) beliefs that a default theory $\langle \mathcal{F}, \mathcal{D} \rangle$ supports. A set of sentences \mathcal{E} is an extension of a default theory $\langle \mathcal{F}, \mathcal{D} \rangle$, if it contains all assumptions that can be consistently added, *and* the inferences that can be made from the first theory \mathcal{F} along with the added assumptions.

$$\sigma \in \mathcal{E} \quad \text{iff} \quad \mathcal{F} \cup \{ \delta \mid \langle \alpha : \beta / \delta \rangle \in \mathcal{D}, \alpha \in \mathcal{E}, \neg\beta \notin \mathcal{E} \} \models \sigma$$

A default rule of the form $\langle \alpha : \delta / \delta \rangle$ is called a normal rule. Normal rules can be used to express rules like “birds fly”,

$$\langle \text{Bird}(\text{tweety}) : \text{Flies}(\text{tweety}) / \text{Flies}(\text{tweety}) \rangle$$

Assume the knowledge base contains only one sentence $\text{Bird}(\text{tweety})$. Here we can see that $\text{Bird}(\text{tweety})$ is already in the extension (the given knowledge base), $\neg\text{Flies}(\text{tweety})$ is not in the extension, and hence $\text{Flies}(\text{tweety})$ can be added to the extension.

Adding some sentences to the extension may preclude the addition of other sentences, which could have been otherwise added. This suggests that a default theory may have more than one extension, indicating that the default theory is ambiguous. Consider the relations depicted in Figure 14.21 on the left. They may be expressed as follows (replacing *CanFly* with *Flies*).

$$\begin{aligned}\mathcal{F}_1 = & \{\text{Penguin(peppy)}, \\ & \forall x(\text{Penguin}(x) \supset \text{Bird}(x)) \\ & \forall x(\text{Penguin}(x) \supset \text{AquaticBird}(x))\}\end{aligned}$$

$$\begin{aligned}\mathcal{D}_1 = & \{ \langle \text{Bird(peppy)} : \text{Flies(peppy)} / \text{Flies(peppy)} \rangle, \\ & \langle \text{AquaticBird(peppy)} : \neg \text{Flies(peppy)} / \neg \text{Flies(peppy)} \rangle \}\end{aligned}$$

There are two extensions for the above default theory⁶,

$$\mathcal{E}_1 = \{\text{Penguin(peppy)}, \text{Bird(peppy)}, \text{AquaticBird(peppy)}, \text{Flies(peppy)}\}$$

and

$$\mathcal{E}_2 = \{\text{Penguin(peppy)}, \text{Bird(peppy)}, \text{AquaticBird(peppy)}, \neg \text{Flies(peppy)}\}$$

In one extension, Peppy can fly and in the other one it cannot. This is to be expected because the theory is inherently ambiguous. However, even when there is a basis for selecting one from two extensions, Default Logic does not have a mechanism to choose it. For example, if instead of saying that “aquatic birds cannot fly” we had asserted “penguins cannot fly” Default Theory still cannot say that the extension in which Peppy the penguin cannot fly, is the preferred extension.

One is compelled to accept one of the two forms of reasoning. A *Skeptical* reasoner accepts a sentence, *only if* it is present in all the extensions. A *Credulous* reasoner accepts a sentence if it is present in *some* extension.

It is also possible that a default theory is inconsistent and has no extension. A simple example of that is when \mathcal{F} is empty, and \mathcal{D} consists of the single rule.

$$\langle \text{true} : \text{Flies(tweety)} / \neg \text{Flies(tweety)} \rangle$$

On the other hand, consider a variation which says it can be assumed that Tweety flies if it is consistent to assume that Tweety flies.

$$\begin{aligned}\mathcal{F}_1 = & \{\forall x(\text{Flies}(x) \supset \text{Alive}(x))\} \\ \mathcal{D}_1 = & \{ \langle \text{Flies(tweety)} : \text{Flies(tweety)} / \text{Flies(tweety)} \rangle \}\end{aligned}$$

This theory has two extensions. The first one contains *Flies(tweety)* and *Alive(tweety)* along with all the tautologies (because they are entailed by anything). In the second one, we only have the tautologies (which are in all extensions anyway). Intuitively, the second one should be preferred and we need to have some notion of minimization. One could for example, discard an extension if a proper subset of it is an extension too. For a more rigorous criterion, the reader is referred to (Reiter, 1980).

Finally, it may be observed that the closed world assumption can be characterized by the rule,

$$\langle \text{true} : \neg P / \neg P \rangle$$

where P is any ground atomic formula.

17.1.4 Autoepistemic Logic

Making default inferences has to necessarily rely on making assumptions in some form. The goal is to be able to exploit associations between categories, for example birds and the ability to fly, without having to commit to the association being a universal rule. In Default Logic, this is done by means of using default rules that dictate when the consequent (and in turn its consequents) can be believed, contingent to some consistency condition. However, these rules are outside the language (FOL) in which knowledge is expressed.

Autoepistemic Logic is designed to bring such rules inside the language, by extending the language with a *modal operator* (Marek and Truszczyński, 1991). The modal operator **B** is intended to take care of the “is consistent” factor in default reasoning.

Given a formula α in first order logic, the phrase **B** α represents belief ⁷ in α . Thus, **B** α stands for the fact that the agent believes α ; \neg **B** α stands for the fact that the agent does not believe α ; **B** $\neg\alpha$ stands for the sentence that the agent believes $\neg\alpha$; and \neg **B** $\neg\alpha$ stands for the sentence that the agent does not believe $\neg\alpha$.

Alternatively, if one is reasoning about the beliefs of an agent, one could treat the formula α to stand for the fact that the agent believes α . In that case, **B** α stands for the fact that the agent believes that it (or she) believes α .

Since these sentences are about the agent’s own beliefs, hence the name autoepistemic logic.

For the purpose of reasoning, we treat **B** as a unary operator. The formulas that are true have no logical basis, except that they have to consistent with what is in the knowledge base. This consistency is enforced by the following set of properties that must be satisfied by the set of beliefs or expansion⁸ \mathcal{K} (see (Brachman and Levesque, 2004)),

1. The set \mathcal{K} is closed under entailment.
If $\mathcal{K} \models \alpha$ then $\alpha \in \mathcal{K}$
2. The set \mathcal{K} respects positive introspection.
If $\alpha \in \mathcal{K}$ then **B** $\alpha \in \mathcal{K}$
3. The set \mathcal{K} respects negative introspection.
If $\alpha \notin \mathcal{K}$ then \neg **B** $\alpha \in \mathcal{K}$

When an expansion satisfies these properties, we say that it is *stable*.

The normal default rule that “birds fly” is expressed in Autoepistemic Logic as,

$$\forall x ((\text{Bird}(x) \wedge \neg \text{B}\neg \text{Flies}(x)) \supset \text{Flies}(x))$$

Now let us say that we have a simple knowledge base consisting of

the single sentence *Bird(tweety)*. The task is to determine the expansions of this knowledge base and check whether *Flies(tweety)* is present in it.

There are two possible expansions of this knowledge base, one in which the statement $\mathbf{B}\neg\textit{Flies(tweety)}$ is *true*, and the other in which it is *false*. The question is whether any of these is stable. Now if $\mathbf{B}\neg\textit{Flies(tweety)}$ is true then $\neg\textit{Flies(tweety)}$ should be present in the expansion, for the second stability criterion to apply. But there is no rule which will add $\neg\textit{Flies(tweety)}$ to the database. Hence, this is not a stable expansion. In the second expansion, $\mathbf{B}\neg\textit{Flies(tweety)}$ is *false* or $\neg\mathbf{B}\neg\textit{Flies(tweety)}$ is *true*. Then by criterion 3 $\neg\textit{Flies(tweety)}$ should not be present in the expansion, which is the case. Now in this expansion, both *Bird(tweety)* and $\neg\mathbf{B}\neg\textit{Flies(tweety)}$ are true and hence *Flies(tweety)* is added to the expansion.

Given a knowledge base *KB*, the set of believable sentences in a stable expansion are only those that are entailed by the formulas that are true in that expansion.

$$\sigma \in \mathcal{E} \quad \text{iff} \quad KB \cup \{\mathbf{B}\alpha \mid \alpha \in \mathcal{E}\} \cup \{\neg\mathbf{B}\alpha \mid \alpha \notin \mathcal{E}\} \models \sigma$$

This suggests an approach to determining whether one might believe a sentence σ or not. The sentence σ can be believed if it is entailed by a stable expansion. The task then is to *search* for a stable expansion by assigning *true* or *false* to each formula of the type $\mathbf{B}\alpha$, and *checking* whether it is stable. The check for stability is done as follows,

- If $\mathbf{B}\alpha$ was replaced by *true* then α should be entailed in the expansion
- If $\mathbf{B}\alpha$ was replaced by *false* then α should not be entailed in the expansion

The algorithm in Figure 17.1 accepts a knowledge base and a query, and returns yes if the query holds under autoepistemic reasoning, and *no* otherwise. For the sake of simplicity, we assume that there is only one level of belief in the knowledge base. That is, there are no formulas of the type $\mathbf{B}\mathbf{B}\alpha$ or $\mathbf{B}\neg\mathbf{B}\alpha$, though they are specified by the stability criteria⁹. In principle, for any formula there are an infinite set of formulas ($\mathbf{B}\alpha$, $\mathbf{B}\mathbf{B}\alpha$, $\mathbf{B}\mathbf{B}\mathbf{B}\alpha$, ...) that should be in the extension, but the level we need is determined by the level of belief that occurs in the *KB*.

Let us consider a knowledge base that contains n occurrences of the **B** operator, $\mathbf{B}\alpha_1$, $\mathbf{B}\alpha_2$, ..., $\mathbf{B}\alpha_n$. We assume that the 2^n combinations of truth values for these formulas are arranged in a total order and a function *AssignBel*(*KB*, i) selects the i^{th} assignment from this ordering. We also assume a function *ApplySimplify*(*assignment*, *KB*) that applies the assignment (*true* or *false*) to the formulas with the **B** operators and simplifies the sentences in which they occur. The function *Stable*(*expansion*, k) tests whether under the k^{th} assignment the expansion is stable. The three functions described above are packed in the procedure *StableExpansion*(*KB*, j) that starts by inspecting the j^{th} combination of truth assignments to the belief operators, and looks at

them one by one in the given ordering, till it finds a stable expansion in which every formula of the type $\mathbf{B}\alpha$ is replaced either by *true* or *false*. The function *Entails* in the main procedure is a sound and complete first order theorem prover. The algorithm given here returns the value determined by the first stable assignment it finds. It can be extended to find all stable assignments.

```

InferenceAE(KB, Alfa)
1  j ← 1
2  while j < 2n
3    ExpNum ← StableExpansion(KB, j)
4    j ← First(ExpNum)+1
5    SA ← Second(ExpNum)
6    if Entails(SA, Alfa)
7      then return "yes"
8  return "no"

StableExpansion(KB, j)
1  for k ← j to 2n
2    A ← AssignBel(KB, k)
3    Ex ← ApplySimplify(A, KB)
4    if Stable(Ex, k)
5      then return (k, Ex)
6  return (k, nil)

```

Figure 17.1 The procedure *StableExpansion* assumes that the combinations of truth assignments to the belief statements are arranged in a total order. It accepts an index j into this ordering and inspects the combinations one by one till it finds a stable expansion. The calling procedure takes this stable expansion and checks whether the formula *Alfa* is entailed in the stable expansion, using a first order theorem prover *Entails*. The functions *First* and *Second* are list functions that return the first and second element respectively. The function *ApplySimplify* substitutes *true* or *false* for the belief statements, as per the chosen combination and simplifies the knowledge base.

The above formulation of searching for a stable expansion adopts a brute force approach which inspects all possible assignments one by one. In practice, one could adopt a procedure like constraint propagation to fix some values for the $\mathbf{B}\alpha_i$ formulas. In particular, if α_i is in the *KB*, only the assignment $\mathbf{B}\alpha_i = \text{true}$ should be considered, and if $\neg\alpha_i$ is in the *KB*, only $\mathbf{B}\alpha_i = \text{false}$ should be considered (because α_i and $\neg\alpha_i$ cannot both be in).

We illustrate the process of searching for stable expansions with a smaller version of *KB*₃.

```

KB5: {  ∀x ∀y (Friend(x,y) ∧ ¬BIsolated(x) ∧ ¬BIsolated(y) ⊃ Chat(x,y))
      Friend(aditi, shubhgata), Friend(aditi, jennifer),
      Isolated(jennifer)
    }

```

We assume that suitable ground instances of the universal formula can be generated. There are three belief statements involved here **BIsolated(aditi)**, **BIsolated(shubhgata)**, and **BIsolated(jennifer)**. Of these, we only consider **BIsolated(jennifer)=true** in the stable expansions because **Isolated(jennifer)** is in the *KB*. We still have four cases to consider.

In the following, a formula of the kind $(\alpha \wedge \text{true} \supset \delta)$ is replaced by $(\alpha \supset \delta)$ and a formula of the kind $(\alpha \wedge \text{false} \supset \delta)$ is removed from the *KB*, since it reduces to *true*.

1. **BIsolated(aditi) = true, BIsolated(shubhgata) = true**

The two implication formulas are,

$$\begin{aligned}
 &(\text{Friend}(\text{aditi}, \text{shubhgata}) \wedge \neg \text{true} \wedge \neg \text{true} \supset \text{Chat}(\text{aditi}, \text{shubhgata})) \equiv \text{true} \\
 &(\text{Friend}(\text{aditi}, \text{jennifer}) \wedge \neg \text{true} \wedge \neg \text{true} \supset \text{Chat}(\text{aditi}, \text{jennifer})) \equiv \text{true}
 \end{aligned}$$

and the resulting knowledge base is

$$KB_{5-1} = \{ \text{Friend}(\text{aditi}, \text{shubhgata}), \text{Friend}(\text{aditi}, \text{Jennifer}), \text{Isolated}(\text{jennifer}) \}$$

This is not a stable expansion since *both Isolated(aditi) and Isolated(shubhgata)* are not entailed, as required by the stability criterion 2.

2. **BIsolated(aditi) = false, BIsolated(shubhgata) = false**

The two implication formulas are,

$$\begin{aligned}
 &(\text{Friend}(\text{aditi}, \text{shubhgata}) \wedge \neg \text{false} \wedge \neg \text{false} \supset \text{Chat}(\text{aditi}, \text{shubhgata})) \\
 &(\text{Friend}(\text{aditi}, \text{jennifer}) \wedge \neg \text{false} \wedge \neg \text{true} \supset \text{Chat}(\text{aditi}, \text{jennifer})) \equiv \text{true}
 \end{aligned}$$

and the resulting knowledge base is

$$KB_{5-2} = \{ \text{Friend}(\text{aditi}, \text{shubhgata}) \supset \text{Chat}(\text{aditi}, \text{shubhgata}), \text{Friend}(\text{aditi}, \text{shubhgata}), \text{Friend}(\text{aditi}, \text{jennifer}), \text{Isolated}(\text{jennifer}) \}$$

This requires that *Isolated(aditi)* and *Isolated(shubhgata)* are not entailed in the *KB*, which is the case. Hence, this is a stable expansion.

And this expansion entails *Chat(aditi, shubhgata)*.

3. **BIsolated(aditi)= true, BIsolated(shubhgata)=false**

The two implication formulas are,

$$\begin{aligned}
 &(\text{Friend}(\text{aditi}, \text{shubhgata}) \wedge \neg \text{true} \wedge \neg \text{false} \supset \text{Chat}(\text{aditi}, \text{shubhgata})) \equiv \text{true} \\
 &(\text{Friend}(\text{aditi}, \text{jennifer}) \wedge \neg \text{true} \wedge \neg \text{true} \supset \text{Chat}(\text{aditi}, \text{jennifer})) \equiv \text{true}
 \end{aligned}$$

and the resulting knowledge base is

$$KB_{5-3} = \{ \text{Friend}(\text{aditi}, \text{shubhgata}), \text{Friend}(\text{aditi}, \text{Jennifer}), \text{Isolated}(\text{jennifer}) \}$$

This is not a stable expansion, since *Isolated(aditi)* is not entailed, as required by the stability criterion 2.

4. **Isolated(aditi)= false, Isolated(shubhagata)=true**

The two implication formulas are,

$$\begin{aligned} (\text{Friend}(\text{aditi}, \text{shubhagata}) \wedge \neg \text{false} \wedge \neg \text{true} \supset \text{Chat}(\text{aditi}, \text{shubhagata})) &\equiv \text{true} \\ (\text{Friend}(\text{aditi}, \text{jennifer}) \wedge \neg \text{false} \wedge \neg \text{true} \supset \text{Chat}(\text{aditi}, \text{jennifer})) &\equiv \text{true} \end{aligned}$$

and the resulting knowledge base is

$$KB_{5-4} = \{\text{Friend}(\text{aditi}, \text{shubhagata}), \text{Friend}(\text{aditi}, \text{jennifer}), \text{Isolated}(\text{jennifer})\}$$

This is not a stable expansion since *Isolated(shubhagata)* is not entailed, as required by the stability criterion 2.

There is a corresponding set of four expansions in which **Isolated(jennifer) = false**. However, all the four are not stable. Therefore, we can see that of the eight possible expansions, only one is stable and in that stable expansion we have the belief that *Chat(aditi, shubhagata)* is true.

17.2 Qualitative Reasoning

Many scientific and engineering applications require us to model and reason about the real world. This reasoning happens in the domain of numerical or *quantitative* data and *quantitative relations* on the data. The world is abstracted into a set of variables, and some kind of a model incorporates the relations between the different variables. For example, the motion of a physical body could be represented by treating it as a point object and defining variables to represent its location, velocity, and acceleration. The speed can be related to a location by a differential equation, as can acceleration be related to speed. Other kinds of models that have been used are finite element analysis, neural networks, constraint systems, linear equations, and so on.

Such quantitative models are an abstraction of the domain. They distill key features of the domain and represent them as mathematical models. Since they are abstractions, they necessarily ignore some aspects of the real world, but this loss of detail¹⁰ is compensated for by being able to reason quickly. Abstraction in any case is inevitable. The degree to which it is done is dictated by the task at hand, the capabilities of the reasoner, and the domain. Newton's Laws of Motion were quite adequate for our day to day reasoning but, as Einstein showed with his Theory of Relativity, are too abstract to deal with motion at very high speeds.

Qualitative models are further abstractions in which one abstracts away from numerical data. Instead of using Hooke's Law for computation, one can express the fact that the force exerted by a spring *increases* as it is stretched away from its normal position. Qualitative Reasoning is also known as Qualitative Physics or Naïve Physics because it has often been used to model physical systems. This is an attempt to replicate the way

people model the everyday world and reason about it very successfully and efficiently.

We illustrate the idea of qualitative reasoning with the following scenario. A young girl is playing with a ball in a room. She releases the ball from a certain location with a certain velocity as shown in Figure 17.2. The task is to predict what will happen next. If we have the numerical data about initial location and velocity, one could apply the Newtonian equations as shown in the figure.

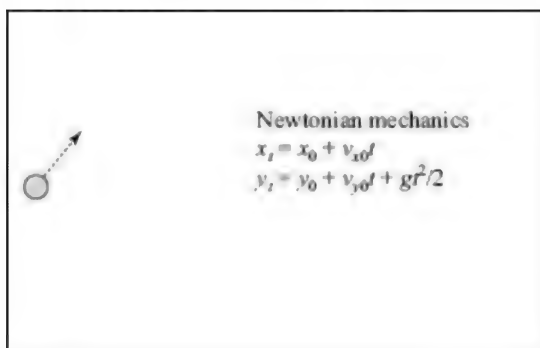


Figure 17.2 Given that a ball is thrown at the location (x_0, y_0) with a velocity (v_{x0}, v_{y0}) , predict the future trajectory of the ball.

Even when we know the initial conditions, there are limitations to using these equations. For one, they are themselves an abstraction of the real world. In particular, they ignore air resistance which has a decelerating effect on velocity. They also ignore the fact that the earth is round, given that the size of the room is miniscule compared to the radius of the earth. More importantly, however, these equations are relations between the different quantities, and are not concerned about the state in which they are applied. One cannot say when they cease to be applicable. That happens in the mind of the reasoner. Such reasoning that factors in extraneous data often happens during the conceptual stage in a design process, before the quantitative analysis kicks in.

Qualitative reasoning treats the moving ball as a being in a *qualitative state*. The state is described as follows. The ball is moving up and to the right, and its vertical speed is decreasing (due to gravity). There are three different ways this state can end, leading to a new state as shown in Figure 17.3. This could happen when the ball hits the ceiling resulting in state 1, or its vertical velocity becomes zero resulting in state 2, or it hits a wall as in state 3. These states are represented by the small black circles. The three different *arcs* themselves represent the identical state, in which the vertical and horizontal velocities are positive, horizontal acceleration is zero, and vertical acceleration is negative.

If one started only with the qualitative information that the ball was moving upwards and sideways under the influence of gravity then one will not be able to predict which of these succeeding states occur. For that,

we would need the quantitative information and perhaps a quantitative model of air resistance as well (imagine doing the same underwater in a swimming pool).

However, a major advantage of using a qualitative model is that even with this limited information, one can chart out all the possibilities that can possibly occur. This is the kind of reasoning an engineer or a scientist might do before reaching out for her calculator. For example, one can safely say that the ball will not start moving leftwards and hit the left wall.

When the ball ceases to be in the moving-up-and-right state, it makes a transition to a different state. Ignoring the instantaneous states 1, 2 and 3 in the above figure, the next possible states succeeding 2 and 3 are shown in solid lines in Figure 17.4.

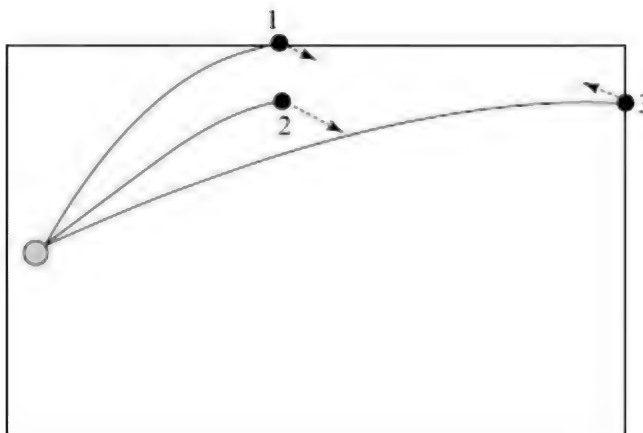


Figure 17.3 In this figure, each of the lines represents a qualitative state – the ball is moving upwards and to the right. A transition can occur either to the state labelled 1 when the ball hits the ceiling, or to state 2 when it stops moving upwards due to gravity, or state 3 when it hits a wall.

As one can see from the figure, after the transition via states 1 and 2, the ball will be in a qualitative state in which it is moving to the right and down. The four solid trajectories in the figure are qualitatively identical states—moving down and right, and will be treated identically by a qualitative reasoning system. For our benefit, we can see that the next transitions will occur in state 1a, 1b, 2a or 2b. In fact, these are only two distinct states. 1a and 2a are identical, when the ball travelling down and right hits a wall. Likewise, 1b and 2b represent the state when it hits the ground. The resulting trajectories are hinted at by the dashed arrows. The states following the transition at 3 are left as an exercise for the reader.

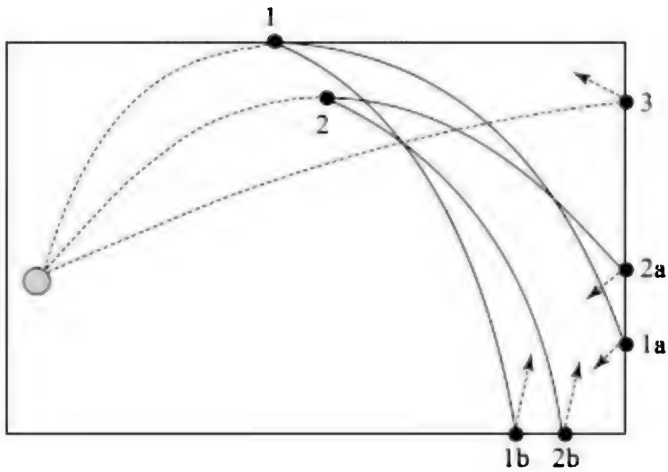


Figure 17.4 State 1 can lead to either state 1a or to state 1b, and likewise for state 2 there are two possible successor states. Observe that 1a and 2a are qualitatively the same state. The successors of state 3 are not shown here.

17.2.1 Representation

The world around us changes continuously. We need finite, symbolic methods for describing continuous change. A qualitative description is one that captures distinctions that make an important, qualitative difference, and ignores the others (Kuipers, 1994).

Consider a person travelling in a train from Chennai to New Delhi on the Tamil Nadu Express. She might have a qualitative notion of her location. She might have some stations on the route as landmarks, and may keep track of her position, only with respect to those landmarks. She might say, for example, that the train is somewhere between Vijayawada and Nagpur. Perhaps breakfast was served at Vijayawada, and lunch is expected at Nagpur. In general, a train passenger may represent its location as being between two stations that it halts at.

The simplest possible variables used for representation are *nominal*, in which the variable either has a value or does not. For example, a mechanic diagnosing a vehicle might say that the engine noise is “normal”, or an enthusiastic trekker might declare that the weather is “clear”. An example that affects us all is the classification of an electric voltage signal into 1 or not 1 (that is, 0) inside our computers.

A more informative type of variable is the *ordinal* type. The different values that the variable can take can be compared according to a base order. These values are typically the landmark values determined by the domain. In the train example above, the stations are ordered being further and further away from the source. Likewise, if we had water being heated in a pan then the landmark values could be the ordered set {absolute zero, freezing point, room temperature, boiling point, infinity}.

Such an ordered set of values defines a *quantity space*. The interval between any two landmarks would be a state, and state change could be *to* or *from* the landmark at the state boundary.

The simplest quantity space contains only three values $\{-, 0, +\}$. In this space, 0 represents the norm, while the deviations are either negative or positive. One can denote this quantity space as $[\bullet]_0$ to represent that the norm is 0. If one wanted to represent the human pulse rate then one could possibly use 72 as the norm.

In the ball, in the room example of Figure 17.2, one could treat the two walls (assuming a two dimensional room), the roof, and the floor as landmarks for positions. The variables and their landmarks would be,

$X\text{-pos}$		$\{\text{Left-wall, Right-wall}\}$
$Y\text{-pos}$		$\{\text{Floor, Roof}\}$
$\delta X\text{-pos} = X\text{-velocity}$		$\{-, 0, +\}$
$\delta Y\text{-pos} = Y\text{-velocity}$		$\{-, 0, +\}$
$\delta X\text{-velocity}$	$= X\text{-acceleration}$	$\{-, 0, +\}$
$\delta Y\text{-velocity}$	$= Y\text{-acceleration}$	$\{-, 0, +\}$

Using quantity spaces for a small number of variables and their derivatives can be surprisingly powerful. Consider a simple system of a pendulum or a spring with a mass. Let its position be represented by a variable X and its velocity by a variable V , both taking values from $[X]_0$ and $[V]_0$. The qualitative behaviour of the pendulum can be depicted by an *envisionment* (de Kleer and Brown, 1984) as shown in Figure 17.5.

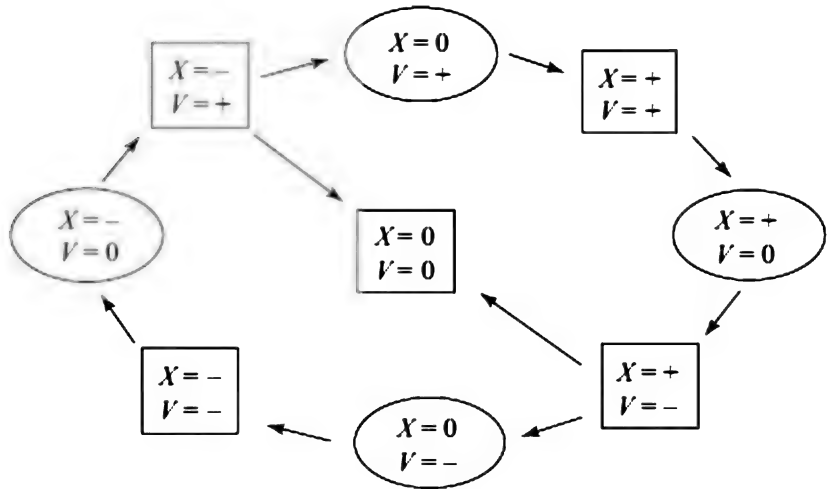


Figure 17.5 The envisionment of all possible trajectories of a simple pendulum. The ovals represent instantaneous states and the rectangles durative states.

Consider the rightmost oval in the figure representing the state in which $[X] = +$ and $[V] = 0$. This could be the state when an agent displaces the pendulum and releases it. It represents an instantaneous

state in which the displacement is positive and velocity is zero. We represent instantaneous states by ovals. The pendulum then moves to a new state in which it has a negative velocity, while displacement remains positive. This is a durative state that persists till the pendulum reaches its resting position ($[X] = 0$), but continues to have a negative velocity. We represent durative states with rectangles.

The oscillatory motion of the pendulum is captured by the movement or evolution of the system state around the circular path in the envisionment. The inner state represents the pendulum at rest, and can be modelled as being reached at the end of the “last” inward swing.

One may want to determine the effect of a set of variables on a given variable. This could be done by operations in the given quantity space similar to addition and multiplication. Addition, denoted \oplus , and multiplication, denoted \otimes , of two variables from the same quantity space can be defined as follows. The addition of two variables is given in Table 17.1.

Table 17.1 Addition over a quantity space

\oplus	$[X] = -$	$[X] = 0$	$[X] = +$
$[Y] = -$	$- \oplus - = -$	$0 \oplus - = -$	
$[Y] = 0$	$- \oplus 0 = -$	$0 \oplus 0 = 0$	$+ \oplus 0 = +$
$[Y] = +$		$0 \oplus + = +$	$+ \oplus + = +$

Observe that addition of a positive value and a negative value is undefined. Consider a water tank with a tap at the bottom to let the water out. Let us say that tank contains some water and the outlet tap is open. We can represent the flow of water from the tap with the variable $[FlowTap] = -$ which says that the flow of water *inwards* is negative. This stands for the fact that water is flowing *out* from the tank. Let us say that at the same time, water is being filled into the tank represented by the variable $[FlowIn] = +$. This represents the fact that water the flow *inwards* is positive. The net flow of water into the tank is $[FlowTap] \oplus [FlowIn]$ which is undefined. This is understandable. Since we only know that water is flowing in as well as flowing out, but do not know the exact rate, we are unable to say whether the net flow is positive or negative or zero.

When the two quantities in question come from *different orders of magnitudes* then it should be possible to decide which one will prevail. For example, if the input flow is *small* and the output flow is *big*, then one can conclude that the net flow is out. One attempt to resolve such ambiguities is to use *order of magnitude* representations, described in the following section.

Multiplication is defined as follows.

Table 17.2 Multiplication over a quantity space.

\otimes	$[X] = -$	$[X] = 0$	$[X] = +$
$[Y] = -$	$- \otimes - = +$	$0 \otimes - = 0$	$+ \otimes - = -$
$[Y] = 0$	$- \otimes 0 = 0$	$0 \otimes 0 = 0$	$+ \otimes 0 = 0$
$[Y] = +$	$- \otimes + = -$	$0 \otimes + = 0$	$+ \otimes + = +$

17.2.2 Orders of Magnitude

Consider the example of a ball hitting the wall of a room. For the sake of argument, let us assume that the wall is moving in a direction towards the ball. The wall can be considered to be part of the earth and, while the ball is in the air, we can think of it as another body. Then, we have a two body collision in which both momentum and energy need to be conserved. But we never think of the earth slowing down as a result of a collision with the ball. This is consistent with orders of magnitude reasoning.

Assuming that there is a completely elastic two body collision, what can we say about the resultant velocities of the two bodies? Given the exact values of the masses and velocities, one can solve the two equations representing the conservation of momentum and conservation of energy. But given only the fact that one has much greater mass than the other, can we make a qualitative prediction? The approach described here was presented by Olivier Raiman (1986; 1991).

Let the two bodies be travelling with linear motion in the opposite direction. Let M_{big} be the mass of the large body and M_{small} be the mass of the small body, and let $M_{\text{big}} \gg M_{\text{small}}$. That is, M_{big} is much much larger than M_{small} . Let $V_i(\text{big})$ and $V_i(\text{small})$ be the initial velocities of the two bodies, and let them be of comparable magnitude and opposite direction. Let $V_f(M)$ and $V_f(m)$ be the final velocities that need to be determined.

When we reason with order of magnitudes, we reason with sets of values rather than individual values. Each set is a quantity space that defines a range of values. Let us say that x and y are two exact values. Instead of knowing the exact values, we might have information about sets X and Y from which these values come from. That is, $x \in X$ and $y \in Y$. We refer to X and Y as *coarse values*. Given such values, we can define the following. Two coarse values are (approximately) equal if the corresponding sets overlap.

$$X \approx Y \text{ iff } X \cap Y \neq \Phi$$

Observe that the \approx operator is not transitive. This implies that one needs to be careful in its use in solving for values. The following operations can be defined on coarse values.

$$X + Y = \{x + y \mid x \in X, y \in Y\}$$

$$XY = \{xy \mid x \in X, y \in Y\}$$

$$-X = \{-x \mid x \in X\}$$

$$X^{-1} = \{x^{-1} \mid x \in X\}$$

Two coarse values can be said to be of different *orders of magnitude*, if they come from or are associated with two different sets called *Small* and *Rough* that satisfy the following properties.

- $0 \in \text{Small}$
- $1 \in \text{Rough}$
- *Small* is closed under addition, multiplication and additive inverse.

$$\text{Small} + \text{Small} = \text{Small}$$

$$\text{Small} \times \text{Small} = \text{Small}$$

$$-\text{Small} = \text{Small}$$

- *Rough* is closed under addition, multiplication, and multiplicative inverse.

$$\text{Rough} + \text{Rough} = \text{Rough}$$

$$\text{Rough} \times \text{Rough} = \text{Rough}$$

$$\text{Rough}^{-1} = \text{Rough}$$

- *Small* absorbs *Rough* for multiplication.

$$\text{Small} \times \text{Rough} = \text{Small}$$

- *Rough* absorbs *Small* for addition

$$\text{Rough} + \text{Small} = \text{Rough}$$

Observe that if you divide a small quantity by a small quantity, one could get a large quantity. Hence, *Small* is not closed under the multiplicative inverse. Likewise, *Rough* is not closed under additive inverse, since one could subtract a large quantity from another large quantity to get a small quantity.

If two nonzero quantities p and q have the same sign, that is $[p]_0 = [q]_0$, and belong to the same coarse set, then their addition results in the same coarse value. That is,

$$\text{IF } [p] = [q]$$

$$\text{THEN}$$

$$\text{AND}$$

$$p \times \text{Small} + q \times \text{Small} = (p+q) \times \text{Small}$$

$$p \times \text{Rough} + q \times \text{Rough} = (p+q) \times \text{Rough}$$

In other words, if we add two small quantities of the same sign, we get another small quantity, and similarly for two rough quantities of the same

size. This has a problem though, because if you keep adding small quantities, eventually the result will be large. Raiman offers a model in which both *Small* and *Rough* are sets of functions over the real interval $(0,1)$, in which each quantity q is mapped onto a real function $f_q(x)$ with x in the open interval. *Small* is the set of functions that tend to 0 as x tends to 1. The set *Rough* has a more involved definition. A function $f_q(t)$ is an element of *Rough*, iff there is a pair of strictly positive reals K^f and K^f , such that as t approaches 1, the function f is bounded below by K^f and bounded above by K^f .

Raiman defines different degrees of “equality” at different *scales* of magnitude, as follows.

1. Id: $p \rightarrow p$ Two quantities are equal on the *Id scale* (only) if they are the same.

2. Close: $p \rightarrow p \times (1 + \text{Small})$ Two quantities are equal on the *Close scale* if they differ by a factor close to 1. If p and q are *Close* then $(p - q)$ would be negligible compared to either of them.

3. Comparable: $p \rightarrow p \times \text{Rough}$ Two quantities are equal on the *Comparable scale* if one can be obtained by multiplying the other with *Rough*. If p and q are *Comparable* and if p is negligible with respect to some t then q would also be negligible with respect to t .

4. Sign: $p \rightarrow p \times \mathfrak{R}^+$ Two quantities are equal on the *Sign scale* if they have the same sign.

Equality on a finer scale implies equality on a coarser scale. For example, if two coarse values are equal on the *Close scale* then they are also equal on the *Comparable scale*. The degree of equality in the *Close* is dependent upon the value of p . One could define an alternative measure called *Near* such that $(p - q) = \text{Small}$.

The relation *Negligible*, denoted \ll , can now be defined on a given scale *Scale* as,

$$(Scale(p) + Scale(q) \approx Scale(q)) \equiv (Scale(p) \ll Scale(q))$$

On the *Close scale*, this translates to,

$$(p \approx q \times \text{Small}) \equiv (Scale(p) \ll Scale(q))$$

We can now express the problem of the two bodies moving towards each other with comparable velocities as follows. Let t stand for the total momentum of the two body system, and V a velocity comparable to both V_{big} and V_{small} .

$$Sign(t) \approx M_{big} V_i(big) + M_{small} V_i(small)$$

$$Close(V_i(big)) \approx Close(V)$$

$$Close(V_i(small)) \approx -Close(V)$$

$$M_{small} \approx M_{big} \times Small$$

As discussed earlier, one has to be careful in using the approximate equality for solving sets of equations. Raiman converts the above relations into the following,

$$0 \subseteq -Sign(t) + M_{big} V_i(big) + M_{small} V_i(small)$$

$$0 \subseteq -Close(V) + Close(V_i(big))$$

$$0 \subseteq Close(V) + Close(V_i(small))$$

$$0 \subseteq -M_{small} + M_{big} \times Small$$

The law of conservation of momentum,

$$M_{big} V_i(big) + M_{small} V_i(small) = M_{big} V_f(big) + M_{small} V_f(small)$$

and the conservation of energy,

$$M_{big} V_i^2(big) + M_{small} V_i^2(small) = M_{big} V_f^2(big) + M_{small} V_f^2(small)$$

The above two yield,

$$V_i(big) + V_f(big) = V_i(small) + V_f(small)$$

Applying a set of inferences too detailed to describe here, Raiman arrives at,

$$V_f(big) \subseteq Close(V)$$

and

$$V_f(small) \subseteq Close(V+V+V)$$

That is, the larger body continues to move approximately with the same velocity V , while the small body bounces back and travels at approximately three times that velocity. The interested reader should verify that this is indeed the case when M_{small} is negligible compared to M_{big} . Raiman's reasoning system, *Estimate*, also shows that if the two masses are *Close* and the velocities are *Close* then the two bodies rebound with the same velocities. If the masses are *Close* and the second body is at rest, or having a comparatively negligible velocity, then the first mass transfers the velocity (or momentum) to the second mass.

17.2.3 Confluences

The orders of magnitude reasoning involves variables, some of which may be negligible as compared to others. Even when such information is not available, interesting inferences can be made by defining relations between different variables that are part of the same physical system. These relations are abstractions of the laws of physics that holds for the system.

Confluences are linear equations over variables that take values from a quantity space. In the following discussion, we use the notation X for a variable and the notation δX for its derivative. A confluence is a summation of such variables that add to a constant. For example, if H is the height of a balloon in the atmosphere, and P is the atmospheric pressure it is subjected to, we could write the confluence,

$$\delta H + \delta P = 0$$

Keeping in mind that these variables come from *Sign* quantity space, we can interpret the confluence as asserting that when the height of the balloon increases ($\delta H = +$), the atmospheric pressure outside it will decrease ($\delta P = -$).

A confluence captures monotonic relations between variables. We could describe Boyle's Law by the confluence $\delta \text{Pressure} + \delta \text{Volume} = 0$ and Charles Law by the confluence $\delta \text{Temperature} - \delta \text{Volume} = 0$. The ideal gas laws would involve all the three variables.

Johann de Kleer demonstrated the use of confluences to model and reason about physical systems. The basic idea is to describe components of systems using confluences. We look at the example presented in (de Kleer and Brown, 1984) in which a pressure regulator is modelled. A hydraulic pressure regulator shown in Figure 17.6 maintains the output pressure at the point labelled OUT in the figure. The basic idea behind the pressure regulator is that if the input pressure (at the point labelled IN) were to increase, it would lead to an increased flow across the valve at a higher pressure, which in turn would push the piston down, thus constricting the opening and reducing the flow and pressure.

The simplest model for this regulator deals with three variables: P the pressure drop across the valve, Q the flow across the valve, and A the cross sectional area regulated by the piston. All the variables take values from the quantity space $[*]_0$.

The following confluence captures the basic relations between the three.

$$\delta P + \delta A - \delta Q = 0$$

One can interpret this confluence in a piecewise fashion. At constant pressure ($\delta P = 0$), as the area available increases so does the flow. To maintain a constant flow ($\delta Q = 0$), one has to increase the pressure drop,

if the area available decreases. In general, a confluence is satisfied if the constituent variables have values consistent with the confluence. The way the regulator works is that as the pressure increases, the flow does too, which in turn pushes the piston down constricting the area, which in turn reduces the flow and the output pressure.

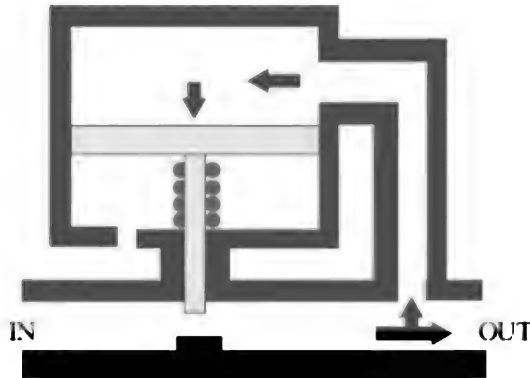


Figure 17.6 A hydraulic pressure regulator.
(Figure adapted from (de Kleer and Brown, 1984)).

The above confluence describes the behaviour only in a qualitative, operational state which we can call the *working* state of the valve. The confluence is not applicable if the valve is either closed or (fully) open. In the former case, no flow takes place and both $[Q]$ and δQ are zero. The input pressure can take any unconstrained value but does not cause a change in the pressure drop. In the latter case, if the valve is fully open the pressure drop across the valve is 0, and cannot change as well. These are different operational states of the valve behaviour and are captured by different sets of confluences. Each state is characterized by a relation on one or more variables, shown in square brackets below. The variables may not be derivatives and the allowed relations are $\{<, \leq, =, \geq, >\}$. A more accurate model then is,

State: open, $[A = A_{\max}]$

State: working, $[0 < A < A_{\max}]$

State: closed, $[A = 0]$

Confluences: $[P] = 0, \delta P = 0$

Confluences: $\delta P + \delta A - \delta Q = 0$

Confluences: $[Q] = 0, \delta Q = 0$

If one wants to allow for the possibility of bidirectional flow across the valve, one must take into account the fact that the pressure drop moves towards zero when the valve opens up with flow in either direction. One could replace the term δA with $[P]\delta A$ to take care of the signs. Equivalently, a *pure* set of confluences is,

State: open, $[A = A_{\max}]$	Confluences: $[P] = 0, \delta P = 0$
State: working ⁺ , $[0 < A < A_{\max}, P > 0]$	Confluences: $[P] = [Q], \delta P + \delta A - \delta Q = 0$
State: working ⁰ , $[0 < A < A_{\max}, P = 0]$	Confluences: $[P] = [Q], \delta P - \delta Q = 0$
State: working ⁻ , $[0 < A < A_{\max}, P < 0]$	Confluences: $[P] = [Q], \delta P - \delta A - \delta Q = 0$
State: closed, $[A = 0]$	Confluences: $[Q] = 0, \delta Q = 0$

17.2.4 ENVISION: Structure to Function

The *envision* system designed by de Kleer and Brown (1984) is designed to reason about physical systems. A physical system is made up of *components* that are drawn from a library of component descriptions, each of which embodies certain behaviour. Each component has a set of *terminals* which allow it to connect to a *conduit*, which in turn is connected to the terminal of another component. Conduits are passive channels used only to transmit material or information between components.

The basic idea behind *ENVISION* is that given that one has knowledge of how each component behaves, and how the components are connected together to form the device, one can infer the behaviour of the device.

Component behaviour + Device structure → Device behaviour

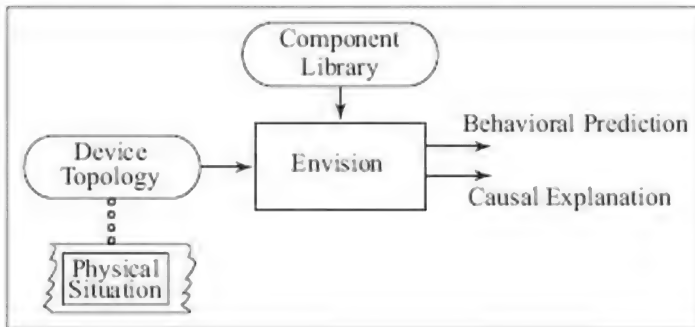


Figure 17.7 *ENVISION*: Component behaviour + device structure → Behaviour.
(Figure adapted from (de Kleer and Brown, 1984)).

For this to be possible, the system has to obey the *no-function-in-structure* principle. The principle says that all the constituent behaviours contributing to the device are generated by the components only, and that the behaviour of each component must be described independent of what it is connected to.

Consider modelling a light switch. A first attempt would model the switch as a relation between its position and the flow of current. When the switch is *on*, the current flows across it, and when it is *off* then it does not. One can connect the switch to a (fluorescent) light bulb and predict that it will emit light when the switch is *on*. However, it is possible that there is a

power cut, and the bulb does not turn on. Or the circuit has two switches in series for some reason. In both these cases, the switch does not behave as expected by its model. The problem, of course, is with the model. It implicitly assumes that there is a potential difference across the switch. A better model would say that if the switch is *on* and there is a voltage drop across its terminals then the current will flow.

A system may breach a qualitative operational state when one of the variables that predicate the state reaches the limit value. *ENVISION* reasons about the possibility of this happening by looking at the derivatives of the corresponding variables. Within an operational state too, variables may attain different combinations of values, which are qualitatively different in nature. For example, in the pressure regulator, if the input pressure goes up then the flow will increase as well, leading to a state in which the piston moves down, in turn reducing the flow. This kind of behaviour can be captured in episodes. Within an episode, a consistent assignment of values needs to be found for the variables in the active confluences.

For example, if pressure regulator is in the Working⁺ state and the pressure starts increasing and δP become positive then for the confluence to be satisfied, either δQ should become positive or δA should become negative or both. Importantly, it disallows a state in which both δQ and δA are zero, or one in which δQ is negative and δA positive. Thus, while it does not allow us to predict what exactly will happen, it does delimit the set of succeeding states to only those that are feasible.

In general, one would like to use constraint propagation (see Chapter 9) to find out what values of other variables are feasible, but it turns out that this is not always possible. Consider the example of two resistances R_1 and R_2 in series as shown in Figure 17.8.

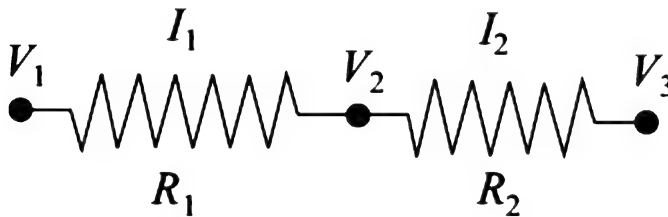


Figure 17.8 Constraint propagation is insufficient to solve for V_2 .

Given the constraints,

$$[R_1] = +$$

$$[R_2] = +$$

$$I_1 = I_2$$

$$I_1 R_1 = V_1 - V_2$$

$$I_2 R_2 = V_2 - V_3$$

and given that $[V_1] = +$ and $[V_3] = 0$

No constraint has enough information to determine the value of any of the unknowns, l_1 , l_2 , or V_2 . At the same time, it is clear to us that the only consistent solution in the domain $\{+,0,-\}$ is $V_2 = +$, $l_1 = +$, and $l_2 = +$. de Kleer calls such systems *inherently simultaneous* because they cannot be solved by propagation of values, and require approaches to solve simultaneous equations. However, given that equality over coarse scales¹¹ has to be handled with care, one cannot carry forward the method of substituting equals for equals. Instead, *ENVISION* resorts to *Generate and Test* based approach, where *Constraint Propagation* is not possible.

We look at the task of predicting the future with an example in the representation scheme introduced by Benjamin Kuipers.

17.2.5 Qualitative Simulation with QSIM

A comprehensive approach to *Qualitative Simulation* is embodied in the algorithm *QSIM* developed by Benjamin Kuipers (1986; 1994). The *QSIM* program, available at the University of Texas at Austin site¹², has proven to be a handy tool for development of QR systems. It extends reasoning over a quantity space with multiple landmark values. Let a variable X over a continuous domain have a set of ordered landmarks $L_0 < L_1 < \dots < L_n$, where L_0 is the minimum value that X can take, and L_n is the maximum value that X can take. *QSIM* represents the qualitative value of the variable with a pair $\langle qmag, qdir \rangle$ where,

$$\begin{aligned} qmag &= L_i \text{ if } X \text{ is on the landmark } L_i \\ &= (L_i, L_{i+1}) \text{ if } X \text{ is between } L_i \text{ and } L_{i+1} \end{aligned}$$

and,

$$\begin{aligned} qdir &= inc \text{ if the value of } X \text{ increasing, that is } \frac{d}{dt}X \text{ is positive} \\ &= std \text{ if the value of } X \text{ steady, that is } \frac{d}{dt}X \text{ is zero} \\ &= dec \text{ if the value of } X \text{ decreasing, that is } \frac{d}{dt}X \text{ is negative} \end{aligned}$$

The initial state of a ball being thrown up in Figure 17.2 would be

$$\begin{aligned} X\text{-pos} &= \langle (\text{Left-wall}, \text{Right-wall}), inc \rangle \\ Y\text{-pos} &= \langle (\text{Floor}, \text{Roof}), inc \rangle \\ X\text{-velocity}^{13} &= \langle (0, \infty), std \rangle \\ Y\text{-velocity} &= \langle (0, \infty), dec \rangle \\ X\text{-acceleration} &= \langle 0, std \rangle \\ Y\text{-acceleration} &= \langle -, std \rangle \end{aligned}$$

QSIM represents relations between variable using constraints. Let

$X(t)$, $Y(t)$ and $Z(t)$ be the values of variables X , Y and Z varying over time t . The variation of variables is assumed to be *reasonable*, that is continuous, with continuous derivatives, and with only a finite number critical points in any bounded interval (Kuipers, 1986). The following relations are defined.

ADD(X, Y, Z) holds over the interval $[a, b]$ iff $X(t) + Y(t) = Z(t)$ for every $t \in [a, b]$

MULT(X, Y, Z) holds over the interval $[a, b]$ iff $X(t)Y(t) = Z(t)$ for every $t \in [a, b]$

MINUS(X, Y) holds over the interval $[a, b]$ iff $X(t) = -Y(t)$ for every $t \in [a, b]$

DERIV(X, Y) holds over the interval $[a, b]$ iff $d/dt X(t) = Y(t)$ for every $t \in [a, b]$

CONSTANT(X) holds over the interval $[a, b]$ iff $d/dt X(t) = 0$ for every $t \in [a, b]$

The relation $X(t) - Y(t) = Z(t)$ would be represented by $Z(t) + Y(t) = X(t)$, and represented as ADD(Z, Y, X).

The value of a variable may be a function of the value of another variable. QSIM allows for qualitative functional relations M^+ and M^- which say that one variable monotonically increases or decreases with the other. This relation can be expressed as a predicate,

$M^+(X, Y)$ holds over the interval $[a, b]$ iff $X(t) = H(Y(t))$ for every $t \in [a, b]$, where H is a function over the domain $Y([a, b])$ and range $X([a, b])$ and $H'(x) > 0$ for x in the interior of its domain.

M^- is defined in an analogous manner, with $H'(x) < 0$. Observe that H stands for a family of functions that are monotonic over the given domain and range.

If at a distinguished time point the different variables related by a particular constraint all have some landmark values, then those values are known as *corresponding values*. For example, consider the amount of pressure exerted by a fluid in a vessel. If the amount (volume) of fluid is zero then the pressure would be zero, and (0,0) would be corresponding values for the two variables.

In the example that follows, we continue to use the direct (mathematical) forms of the above constraints for the sake of readability. Consider an example in which there are two tanks, A and B , connected by a pipe at the base, as shown in Figure 17.9¹⁴. The capacity of tank A is $AMAX$ and tank B is $BMAX$. In the qualitative reasoning that follows, we do not need the actual numerical values. We assume that the connecting pipe is a conduit that does not have any effect of its own, and that the flow of the fluid is such that momentum has no impact. This could happen for example if the fluid has high viscosity.

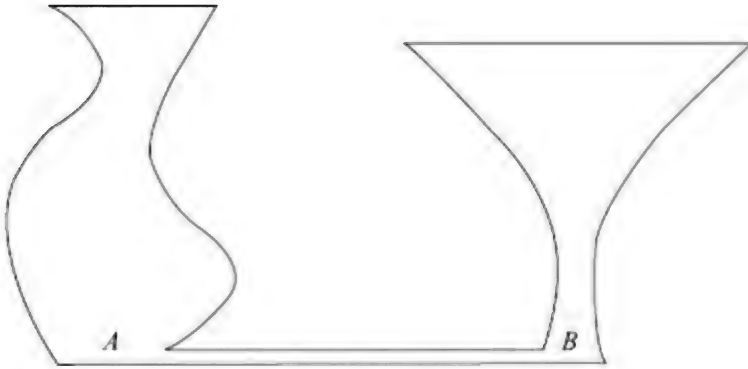


Figure 17.9 Two tanks connected with a pipe at the base.

The model for the two tanks is based on relations on the following qualitative variables shown with their quantity spaces.

Table 17.3 The variables in the 2-tank system

<i>Variable name</i>	<i>Quantity space</i>	<i>Description</i>
<i>amtA</i>	$(0, A_{MAX}, \infty)$	Amount (volume) of fluid in tank <i>A</i>
<i>amtB</i>	$(0, B_{MAX}, \infty)$	Amount (volume) of fluid in tank <i>B</i>
<i>total</i>	$(0, \infty)$	$amtA + amtB$
<i>pressureA</i>	$(0, \infty)$	Pressure at the base of tank <i>A</i>
<i>pressureB</i>	$(0, \infty)$	Pressure at the base of tank <i>B</i>
<i>pAB</i>	$(-\infty, 0, \infty)$	Pressure drop from <i>A</i> to <i>B</i>
<i>flowAB</i>	$(-\infty, 0, \infty)$	Flow from <i>A</i> to <i>B</i>

Observe that the quantity spaces above, capture only partial knowledge. One could introduce more landmarks, for example a value *maxPressureA* for the *pressureA* variable, and likewise for *pressureB*, *pAB*, and *flowAB*. However, these are enough for the reasoning to start with, and QSIM allows for more landmarks to be discovered on the way. More significantly, these landmark values do not have to be numeric quantities. Symbolic ones will do.

The following constraints describe the relations between the different variables.

- The pressure in each tank is proportional to the volume of water in the tank.

$$pressureA = M^+(amtA)$$

$$pressureB = M^+(amtB)$$

- The pressure drop across the conduit is equal to the difference in pressures in the two tanks.

$$p_{AB} = pressureA - pressureB$$

- The rate of flow rate across the conduit pipe is proportional to the pressure drop between the two tanks.

$$flow_{AB} = M^*(p_{AB})$$

- The flow rate across the pipe equals the increasing amount in one tank and decreasing amount in the other.

$$\frac{d}{dt}amtB = flow_{AB}$$

$$\frac{d}{dt}amtA = -flow_{AB}$$

- The amount of fluid in the two tanks is conserved. If flow occurs from one tank to the other then the amount decreases in the former and increases in the latter.

$$amtA + amtB = \text{total Constant}(\text{total})$$

The set of constraints can be represented by the following diagram adopted from (Kuipers, 1994). Observe that the M^* constraint is a symmetric constraint. Thus, when $pressureB = M^*(amtB)$, it is also the case that $amtB = M^*(pressureB)$. The circle alongside the $\frac{d}{dt}$ on the left in the figure represents negation.

Let us take a simple situation in which the two tanks are empty, and fluid is added to tank A till it becomes full. We can assume that this fluid is added instantaneously, or alternatively we can assume that there is a valve in the pipe that is opened after the fluid is added. What can one say about the events that will happen next?

The available information is a partial description of the state. At time $t = t_0$, the following partial information is known.

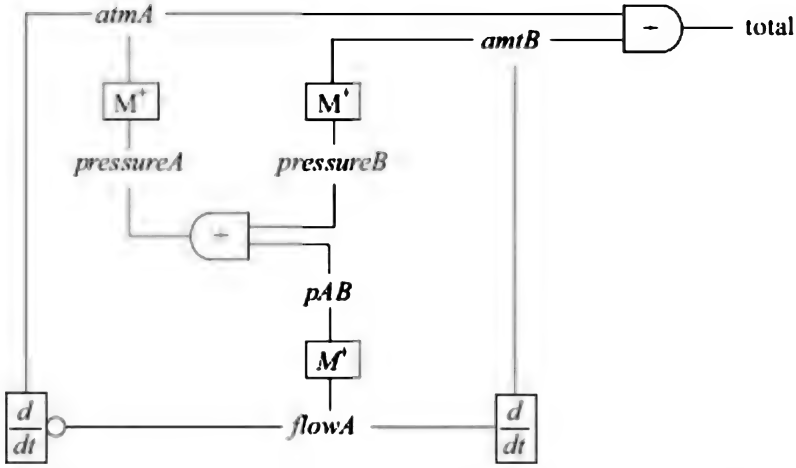


Figure 17.10 The constraint diagram for the 2-tank system. The constraints are in lined “boxes”.

$$\begin{array}{ll}
 amtA = \langle AMAX, ? \rangle & - \quad \text{tank } A \text{ is full} \\
 amtB = \langle 0, ? \rangle & - \quad \text{tank } A \text{ is empty}
 \end{array}$$

The following propagation of constraints yields more information about the given state at time $t = t_0$ (Kuipers, 1994).

1. Because of the corresponding value at $(0,0)$,

$$amtB.qmag = 0 \Rightarrow pressureB.qmag = 0$$

2. Because the corresponding values at $(0,0)$ and (∞, ∞) , we can conclude that

$$amtA.qmag = AMAX \Rightarrow pressureA.qmag = (0, \infty)$$

Note that because of the way the quantity space has been constructed for *pressureA*, this is the best description we can get. A more detailed model would have something like *maxPressureA*, a corresponding value. Given that pressure has only two landmarks in our description, it can have only two corresponding values, and since the system is at neither, it follows that pressure must be somewhere between those landmark values.

3. An addition constraint has an implicit set of corresponding values at $(0,0,0)$. If one of the constituents is not zero, and the other is zero then the sum must be nonzero.

$$amtA.qmag = AMAX \wedge amtB.qmag = 0 \Rightarrow total.qmag = (0, \infty)$$

and likewise

$$pressureA.qmag = (0, \infty) \wedge pressureB.qmag = 0 \Rightarrow pAB.qmag = (0, \infty)$$

4. Constant values, by definition, have no direction of change (i.e. std)

$$\text{Constant}(\text{total}) \Rightarrow [d/dt \text{ total}] = 0 \text{ or } \text{total}.qdir = \text{std}$$

5. The corresponding values at (0,0) and (∞, ∞) of the constraint for *flowAB* and *pAB* imply that since *pAb* is at neither, so *must flowAB* be.

$$pAB.qmag = (0, \infty) \Rightarrow flowAB.qmag = (0, \infty)$$

6. The derivative constraints can now determine the direction of change of *amtA* and *amtB*.

$$flowAB = (0, \infty) \Rightarrow [d/dt(\text{amtA})] = - \wedge [d/dt(\text{amtB})] = +$$

or

$$\text{amtA}.qdir = \text{dec} \wedge \text{amtB}.qdir = \text{inc}$$

The known direction of change of *amtA*, *amtB* and *total* is consistent with the addition constraint.

7. The direction of changes propagate through monotonic functions

$$\text{amtA}.qdir = \text{dec} \Rightarrow \text{pressureA}.qdir = \text{dec} \text{ and } \text{amtB}.qdir = \text{inc} \Rightarrow \text{pressureB}.qdir = \text{inc}$$

8. Direction of change *can* also propagate through the addition constraint,

$$\text{pressureA}.qdir = \text{dec} \wedge \text{pressureB}.qdir = \text{inc} \Rightarrow pAB.qdir = \text{dec}$$

9. And finally, the direction of change in pressure drop propagates to flow.

$$pAB.qdir = \text{dec} \Rightarrow flowAB.qdir = \text{dec}$$

Given that we started with the partially known state in which tank A has *amtA* amount of fluid and tank B is empty, reasoning over the constraint model gives us a better description of the state. In this case, it is a complete description as depicted in Table 17.4.

Table 17.4 The initial state with tank A is partially filled

Variable name	qmag	qdir	Quantity space	Target value
<i>amtA</i>	<i>AMAX</i>	<i>dec</i>	$(0, AMAX, \infty)$	$(0, AMAX)$
<i>amtB</i>	0	<i>inc</i>	$(0, BMAX, \infty)$	$(0, BMAX)$
<i>total</i>	$(0, \infty)$	<i>std</i>	$(0, \infty)$	—
<i>pressureA</i>	$(0, \infty)$	<i>dec</i>	$(0, \infty)$	0
<i>pressure</i>	0	<i>inc</i>	$(0, \infty)$	$(0, \infty)$
<i>pAB</i>	$(0, \infty)$	<i>dec</i>	$(-\infty, 0, \infty)$	0
<i>flowAB</i>	$(0, \infty)$	<i>dec</i>	$(-\infty, 0, \infty)$	0

The qualitative state described above exists only for an instant. There are three variables—*amtA*, *amtB*, and *pressureB*—on landmark values, and all three are changing. All three move simultaneously away from their landmarks and the state transitions to a durative state that exists over a open interval (t_0, t_1) . The new state is described in Table 17.5 below.

Table 17.5 The first qualitative state in the next instant

<i>Variable name</i>	<i>qmag</i>	<i>qdir</i>	<i>Quantity space</i>	<i>Target value</i>
<i>amtA</i>	$(0, AMAX)$	<i>dec</i>	$(0, AMAX, \infty)$	0
<i>amtB</i>	$(0, BMAX)$	<i>inc</i>	$(0, BMAX, \infty)$	<i>BMAX</i>
<i>total</i>	$(0, \infty)$	<i>std</i>	$(0, \infty)$	—
<i>pressureA</i>	$(0, \infty)$	<i>dec</i>	$(0, \infty)$	0
<i>pressureB</i>	$(0, \infty)$	<i>inc</i>	$(0, \infty)$	∞
<i>pAB</i>	$(0, \infty)$	<i>dec</i>	$(-\infty, 0, \infty)$	0
<i>flowAB</i>	$(0, \infty)$	<i>dec</i>	$(-\infty, 0, \infty)$	0

During this qualitative state that exists till the yet unknown time point t_1 , fluid is flowing from tank *A* to tank *B*. In that sense, the physical state is changing even when qualitatively it remains the same. The qualitative state will change when one of the variables reaches a different qualitative value. This could happen when the magnitude *qmag* reaches a landmark in the direction it is moving to. This could also happen if *qdir* changes. That is, it stops moving (or starts moving if it was steady). In addition, different variables could change at the same time.

In the above problem, the variable *amtA* is moving towards 0. It could reach that landmark (at least when seen in isolation) or it could stop decreasing (if *qdir* becomes *std*). Likewise, *amtB* could reach *BMAX*, or stop increasing. The variable *pressureA* could reach 0, but its value is correlated to *amtA*, and if it happens, both will reach 0 at the same instant. Its counterpart *pressureB* is headed towards ∞ , and one could always exclude that from the set of possibilities. Finally *pAB* could move towards 0. Observe that if that happens, then *pressureA* would become equal to *pressureB*, the flow *flowAB* would reach 0 as well, and the variables *pressureA*, *pressureB*, *pAB*, *flowAB*, *amtA* and *amtB* would all become steady.

There are three distinct possibilities and their combinations left.

1. *amtA* → 0 This implies *pressureA* → 0. This in turn implies that *pAB* → $(-\infty, 0)$. But that is not possible without going through the qualitative state in which *pAB*=0, a landmark it is headed to. So this possibility can be excluded.

2. *pAB* → 0 As discussed above, all variables move to a steady state.

3. *amtB* → *BMAX* At this time point, if *pAB* ≠ 0, the fluid overflows the tank (*amtB* → $\langle BMAX, std \rangle$), *amtA* = $\langle (0, AMAX), dec \rangle$ and as a consequence *total.qdir* → *dec*. Thus, the constraint *Constant(total)* does not hold

anymore and the model breaks down. And the reasoner can recognize this fact.

The three distinct *feasible* successor states therefore may be reached when,

1. $pAB \rightarrow 0$
2. $pAB \rightarrow 0$ and $amtB \rightarrow BMAX$
3. $amtB \rightarrow BMAX$, in which case the constraint model of Figure 17.10 no longer applies and the system moves to a different operational state.

The qualitative state reached in case 1 (and also case 2 with a minor change) is,

Table 17.6 A steady qualitative state at time t_1

Variable name	qmag	qdir	Quantity space	Target value
amtA	(0, AMAX)	std	(0, AMAX, ∞)	—
amtB	(0, BMAX)	std	(0, BMAX, ∞)	—
total	(0, ∞)	std	(0, ∞)	—
pressureA	(0, ∞)	std	(0, ∞)	—
pressure	(0, ∞)	std	(0, ∞)	—
pAB	0	std	($-\infty$, 0, ∞)	—
flowAB	0	std	($-\infty$, 0, ∞)	—

Such a steady state could be used to define new landmark values for the different variables, giving us the following refined description,

Table 17.7 New landmark values defined at time t_1

Variable name	qmag	qdir	Quantity space	Target value
amtA	ASTD	std	(0, ASTD, AMAX, ∞)	—
amtB	BSTD	std	(0, BSTD, BMAX, ∞)	—
total	TSTD	std	(0, TSTD, ∞)	—
pressureA	PASTD	std	(0, PASTD, ∞)	—
pressure	PBSTD	std	(0, PBSTD, ∞)	—
pAB	0	std	($-\infty$, 0, ∞)	—
flowAB	0	std	($-\infty$, 0, ∞)	—

A new set of correspondences is also formed as a result of these landmarks. For, $amtA=ASTD$ and $pressureA=PASTD$ forms the correspondence (ASTD, PASTD). The others are left as an exercise for the reader.

The above example illustrates the kind of predictions that qualitative reasoning allows us to make in the face of uncertain information. Given that we do not know the capacities of the two tanks, the best we can say is that one of the three cases will happen. A steady state could be reached with either the tank *B* partially full, or filled to the brim, or the system could move to a different operational qualitative state in which tank *B* overflows, and the total fluid between the two tanks reduces in amount. And only one of these behaviours can be demonstrated by the

system. The actual one that happens can only be determined with the knowledge of the capacities of the two tanks.

Qualitative reasoning is thus a powerful tool to reason about a system without resorting to numerical information. This is the kind of common sense reasoning that people do before without going into quantitative analysis.

17.3 Model Based Diagnosis

The ability to construct a model of a system and generate predictions based on the model can be considered to be deep knowledge of the system. This ability can be exploited in other ways as well. In this section, we look at how this can lead to an approach to diagnosis of (malfunctioning) systems.

The earliest diagnosis systems were based on experiential knowledge. Such knowledge could be in the form of rules, as described in Chapter 6. The rules embody a distilled and modular form of knowledge that has to be elicited from a domain expert. It is assumed that the domain expert would be able to delve into her vast experience and mine the rules from her memory. This however was a somewhat unexpected bottleneck (see for example (Feigenbaum, 1977), (Forsythe and Buchanan, 1989)) in building systems that would exploit expert knowledge. It turns out that the experts were either unable or unwilling to articulate such knowledge that would be the core of a rule based expert system.

To a certain extent, this led to the evolution of Case Based Reasoning approaches, described in Chapter 15, in which the strategy is to capture entire problem solving experiences, and retrieve the best matching one when a new problem occurs. In diagnosis, the problem is the description of abnormal behaviour of a device, or a living creature, and the solution is a diagnosis and a therapy. Numerous helpdesk applications were built in which a relatively inexperienced call centre employee would respond to a user, in consultation with a CBR system (see (Watson, 1997; 2002) for case studies).

The problem with experience based systems is that they apply only to those problems for which knowledge has been gleaned from experience. For diagnosis of devices, that means that only those devices for which such experience has accrued can be diagnosed. A new device may have to wait till expert human users have provided the solutions. In that sense, experience based diagnosis is device specific.

Model based diagnosis adopts a more fundamental approach. The basic idea is that if one can predict the (expected) behaviour of a system with a model, and if the observed behaviour is different from the expected one, then one knows that there is a fault. Further, if the predicted behaviour in the model is derived from the known behaviour of the components and the known structure of the device then the discrepancy

between expected and observed behaviours can be used to drive reasoning algorithms to determine which components are faulty. This approach to diagnosis is also known as *Consistency Based Diagnosis* (Reiter, 1987).

Since model based diagnosis works with an explicit declarative model of the system, we also say that it relies on *deep* knowledge. Such a model could be constructed during the design phase of the device. Then the behaviour of each component can be stored in a component library, and structure of the device would define the expected behaviour.

The diagnosis algorithms themselves can be developed in a device independent fashion. This is in line with the general approach of knowledge based systems in which the knowledge itself exists in a declarative form, and can possibly be used in different tasks. The diagnosis algorithms are themselves domain independent. The following figure, adapted from (Struss and Price, 2004), depicts how a device specific diagnosis system could be constructed using the three knowledge sources—library of component behaviours, the structure information from CAD, and the general diagnosis algorithm.

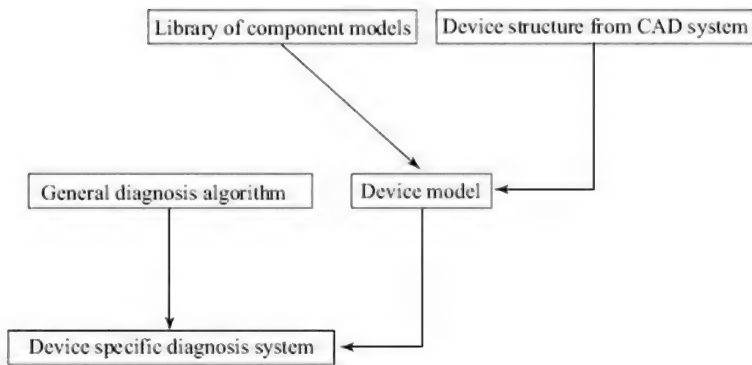


Figure 17.11 A generic approach to building model based diagnosis systems.

17.3.1 Component Models

A physical device in operation receives one or more inputs and produces one or more outputs. Given a device, a fault is said to occur if the observed behaviour (or output) is not an expected one.

The device itself is made up of a set of (active) components connected in a known way via a set of (passive) conduits. We define the task of diagnosis as follows. Given that a fault has occurred, the task is to identify the set of components that are broken (that is, not working as designed). This is also known as *fault localization*. Observe that we confine ourselves to searching for components being faulty. The conduits and terminals are assumed to be passive. This means that if a wire connecting two electronic components or a pipe in a hydraulic system is

to be included in the purview of the diagnosis process then it too should be modelled as a component. The behaviour of the component could simply be to transmit voltage or fluid from one end to another, and a broken component would not do so.

Observe also that we do not consider the possibility of the structure—the choice of components and the way they are connected—as being faulty. We assume that the design of the system is correct. However, this line of reasoning has been suggested for the task of redesign of products (see (Bakker et al., 1994), (Stumptner and Wotawa, 1998)). The idea here is to treat the design problem as a redesign problem, begin with a (faulty) design, and apply model based approaches to arrive a design to match the new specifications.

Figure 17.12 shows an example which has been commonly used in the model based diagnosis literature. The device consists of three multipliers Mul_1 , Mul_2 and Mul_3 , which feed their output to two adders Add_1 and Add_2 , whose output appears at terminals F_1 and F_2 respectively. The figure shows the values of the six inputs and the expected outputs at F_1 and F_2 . The actual output for Add_1 shown in square brackets is at variance with the expected one. A problem of diagnosis is at hand.

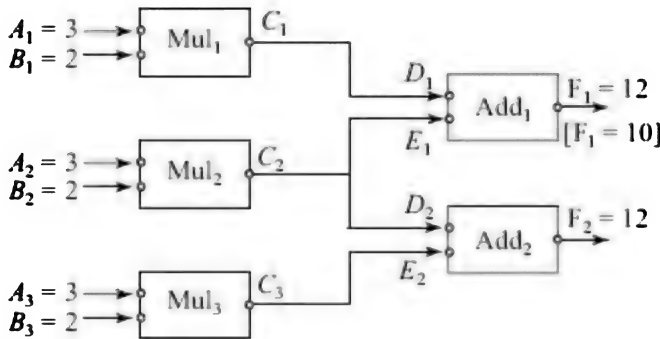


Figure 17.12 A simple device made of three multipliers and two adders. A fault has occurred because the observed value at $F_1 = 10$ differs from the predicted value 12.

A model for the above device should allow us to compute the expected output, given any input. For example, the components could be modelled as functions in a First Order Logic (FOL) with equality (see Chapter 12).

$$\begin{aligned} \text{Multiplier}(M) &\equiv \text{output}(M) = \text{Product}(\text{input}_1(M), \text{input}_2(M)) \\ \text{Adder}(A) &\equiv \text{output}(A) = \text{Sum}(\text{input}_1(A), \text{input}_2(A)) \end{aligned}$$

The structure of the device shown above could be modelled as,

$$\begin{aligned}
& \text{Multiplier}(\text{Mul}_1) \wedge \text{Multiplier}(\text{Mul}_2) \wedge \text{Multiplier}(\text{Mul}_3) \wedge \text{Adder}(\text{Add}_1) \wedge \\
& \text{Adder}(\text{Add}_2) \wedge \text{input}_1(\text{Mul}_1) = A_1 \wedge \text{input}_2(\text{Mul}_1) = B_1 \wedge \text{input}_1(\text{Mul}_2) = A_2 \wedge \text{input}_2(\text{Mul}_2) = B_2 \\
& \wedge \text{input}_1(\text{Mul}_3) = A_3 \wedge \text{input}_2(\text{Mul}_3) = B_3 \wedge \text{input}_1(\text{Add}_1) = D_1 \wedge \text{input}_2(\text{Add}_1) = E_1 \\
& \wedge \text{input}_1(\text{Add}_2) = D_2 \wedge \text{input}_2(\text{Add}_2) = E_2 \\
& \wedge \text{output}(\text{Mul}_1) = C_1 \wedge \text{output}(\text{Mul}_2) = C_2 \wedge \text{output}(\text{Mul}_3) = C_3 \\
& \wedge \text{output}(\text{Add}_1) = F_1 \wedge \text{output}(\text{Add}_2) = F_2 \\
& \wedge C_1 = D_1 \wedge C_2 = E_1 \wedge C_2 = D_2 \wedge C_3 = D_2
\end{aligned}$$

The sentence $C_1 = D_1$ says that the output of Mul_1 is connected to the input of Add_1 , and likewise for similar statements. The reader should verify that given the facts $A_1 = 3$, $B_1 = 2$, $A_2 = 3$, $B_2 = 2$, $A_3 = 3$, and $B_3 = 2$, the values $F_1 = 12$ and $F_2 = 12$ follow.

However, as pointed out by Davis and Hamscher (1988), being able to reason in the forward direction is not enough. The task of diagnosis requires that one should be able to propagate the observed discrepancy to the various connected components in the device. This means that one should be able to infer values at hidden terminals. For example, if one knew that $E_1=6$ and the observed value $F_1=10$, then one should be able to say that, as long as the component Add_1 is working correctly, the value at D_1 should be 4. To facilitate this, the inputs and outputs of a component should be expressed by a set of constraints or as relations that allow inferences for any unknown variable in terms of the others. For example, we could describe the multiplier and adder using predicates in logic,

$$\begin{aligned}
\text{Multiplier}(M) &\equiv \text{Product}(\text{output}(M), \text{input}_1(M), \text{input}_2(M)) \\
\text{Adder}(A) &\equiv \text{Sum}(\text{output}(A), \text{input}_1(A), \text{input}_2(A))
\end{aligned}$$

where $\text{Product}(X, Y, Z)$ is *true* iff $X = YZ$ and $\text{Sum}(X, Y, Z)$ is *true* iff $X = Y + Z$. Or one could use a constraint representing language along with a constraint solver.

17.3.2 Consistency Based Diagnosis

Given a device model in FOL, diagnosis can be arrived at by a process of logical reasoning, called Consistency Based Diagnosis, formalized as follows by Raymond Reiter (1987).

From the device designer's perspective, each component can be described by its *intended behaviour*. In the above example we can say,

$$\begin{aligned}
\text{Multiplier}(M) &\supset \text{Product}(\text{output}(M), \text{input}_1(M), \text{input}_2(M)) \\
\text{Adder}(A) &\supset \text{Sum}(\text{output}(A), \text{input}_1(A), \text{input}_2(A))
\end{aligned}$$

This description can be thought as *default* behaviour, or the behaviour expected in general when the device is working as designed. For the task of diagnosis, the reasoner may have to contend with the situation when some component is not working as designed. To model the device to cater to reasoning about malfunctioning components, we assume that each component in the device is either working or broken. Reiter adopts

the predicate $Ab(x)$ introduced by McCarthy (see Section 17.1.2) to indicate that a component is abnormal or broken. The designed behaviour of a component is as expected, only as long as the component is not broken. For the adder and the multiplier we can write,

$$\begin{aligned} Multiplier(M) \wedge \neg Ab(M) &\supset Product(output(M), input_1(M), input_2(M)) \\ Adder(A) \neg Ab(A) &\supset Sum(output(A), input_1(A), input_2(A)) \end{aligned}$$

As a corollary, one can infer that if the consequent in the implication is false, then the antecedent must be false too. Assuming that we continue to call a broken component by its category name, one can deduce that it is abnormal (or broken). In the multiplier example above, it means $Ab(M)$ is *true*¹⁵.

Let SD be the system description of the device. SD is the set of FOL sentences as described above. This includes the component definitions and also the structure description. Let $COMP$ be the set of components in the system. Let MA stand for mode assignment, as described in (Struss, 2008), describing the status of each component. Let MA_{OK} be the mode assignment when the system is functioned correctly (as expected or designed). Then for each component $c_i \in COMP$, we have a sentence $\neg Ab(c_i)$ in MA_{OK} asserting that c_i is okay. Let OBS be a set of observations about a system.

When the system works as expected, the set of sentences $SD \cup OBS$ is consistent. That is, one cannot derive a contradiction (represented by \perp) from them.

$$SD \cup MA_{OK} \cup OBS \not\vdash \perp$$

This happens for example when,

$$OBS = \{A_1 = 3, B_1 = 2, A_2 = 3, B_2 = 2, A_3 = 3, B_3 = 2, F_1 = 12, F_2 = 12\}$$

However, if the set of observations instead are,

$$OBS = \{A_1 = 3, B_1 = 2, A_2 = 3, B_2 = 2, A_3 = 3, B_3 = 2, F_1 = 10, F_2 = 12\}$$

where $F_1=10$ is a different value, then the system description SD and observations OBS are no longer consistent. That is,

$$SD \cup MA_{OK} \cup OBS \vdash \perp$$

This implies that there is some statement in $SD \cup MA_{OK}$ that is leading to the contradiction. This can only be in MA_{OK} , since we assume that the system *design* is correct. The task of diagnosis then is to find a new mode assignment MA_X which is consistent with the observations. That is,

$$SD \cup MA_X \cup OBS \not\vdash \perp$$

Let us take a one component system which is a multiplier, whose inputs are 4 and 7 and the output reads 20. Then,

$$\begin{aligned} SD &= \{Multiplier(M), \\ &\quad Multiplier(M) \wedge \neg Ab(M) \supset Product(output(M), input_1(M), input_2(M))\} \\ MA_{OK} &= \{\neg Ab(M)\} \\ OBS &= \{input_1(M) = 4, input_2(M) = 7, output(M) = 20\} \end{aligned}$$

It follows that,

$$Product(20, 4, 7)$$

which, we know from our knowledge of arithmetic, is a contradiction or a false statement. Since the only statement we can retract is $\neg Ab(M)$, we are compelled to assert $Ab(M)$. That is, the multiplier M has been identified as a faulty component. Since the left hand side of the implication is no longer true, the following set of statements is consistent (does not lead to a contradiction).

$$\begin{aligned} SD &= \{Multiplier(M), \\ &\quad Multiplier(M) \wedge \neg Ab(M) \supset Product(output(M), input_1(M), input_2(M))\} \\ MA_1 &= \{Ab(M)\} \\ OBS &= \{input_1(M) = 4, input_2(M) = 7, output(M) = 20\} \end{aligned}$$

In the case of a more complex device with say N components, we have $2^N - 1$ different possible mode assignment sets MA_X from which to pick one that makes the overall set of sentences consistent.

One extreme set is the one in which we assert that *all* the N components are broken. Clearly, this will make the set of sentences consistent. However, that would not serve any purpose, because our task is to *identify* which component is faulty, or which components are faulty. The principle of parsimony (Reiter, 1987) says that “a *diagnosis is a conjecture that some minimal set of components are faulty*”.

Given that an N -component device is faulty, the task of diagnosis then is to search through various subsets of components to identify a minimal set, which if assumed to be broken would explain, or be consistent with, the observations.

Given an observation OBS , Reiter defines the notion of a *Conflict Set* (CS) as a set of components of which at least one must be broken. That is, it is the set of suspects that could be causing the observed discrepancy. We can then define a conflict set as follows.

A set $CS \subseteq COMP$ is a conflict set with respect to an observation OBS iff,

$$SD \cup \{\neg Ab(c) \mid c \in CS\} \cup OBS \vdash \perp$$

Clearly, a superset of a conflict set would also be a conflict set. A *minimal* conflict set is a conflict set that does not have a proper subset that is a conflict set. If one has a collection of minimal conflict sets, then

one would need to pick a *hitting set*, such that every conflict set has at least one member in the hitting set. Such a hitting set would be a candidate diagnosis if it does not have a proper subset that is a hitting set.

As Reiter has observed in his seminal paper, the task of diagnosis can be seen as a task of default reasoning. Seen from the perspective of *Circumscription* (see Section 17.1.2), given a system description and a set of observations, the task is to find a minimal set of $Ab(x)$ predicates such that the set of sentences is consistent. A set of components $\Delta \in COMP$ is a diagnosis, if Δ is a *minimal* set such that,

$$SD \cup OBS \cup \{\neg Ab(c) \mid c \in COMP \setminus \Delta\}$$

is consistent. In other words,

$$SD \cup OBS \cup \{\neg Ab(c) \mid c \in COMP \setminus \Delta\} \not\models \perp$$

A brute force approach would try the different subsets in increasing order of size. However, given that we know which of the output terminals has a discrepancy, and know the connection topology of the device, one can surely do better than search blindly.

17.3.3 The General Diagnostic Engine

Very often, one observation may not be enough to pinpoint the fault. One may have to generate more observations. This could be done either by changing the input parameters if possible, or by measuring values of internal terminals where possible.

For each observation OBS , one can find a set of minimal conflicts¹⁶ denoted $\{CS\}_{OBS}$. As defined above, each conflict contains a group of suspects, at least one of which must be broken. Let $\{OBS\}$ be the set of all observations and $\{CS\}$ the union of all conflict sets associated with the different observations. Observe that $\{CS\}$ grows monotonically with the observations.

A conflict set identifies a set of components, at least one of which is broken. A hitting set $HS_{\{CS\}}$ covers each conflict set, in the sense that it identifies at least one element in it. A hitting set HS signifies the fact that *all* components in the hitting set are broken. That is, we associate a mode assignment MA_{HS} with the hitting set that assigns a broken status to all components in the hitting set, and okay to the remaining components. A hitting set HS is minimal if no proper subset of HS is a hitting set. A minimal hitting set satisfies the following property, and is therefore a candidate for being a diagnosis.

$$MA_{HS} = \{Ab(c) \mid c \in HS\} \cup \{\neg Ab(c) \mid c \in COMP \setminus HS\}$$

and

$$SD \cup MA_{HS} \cup OBS_X \not\models \perp \quad \text{for all observations } OBS_X \in \{OBS\}$$

Let $\{HS\}$ be the set of hitting sets associated with $\{CS\}$. Each hitting set HS in $\{HS\}$ is a candidate diagnosis, and says that every component in HS is broken. Also observe that,

$$\{HS\} \subseteq 2^{COMP}$$

The General Diagnostic Engine (GDE) presented by de Kleer and Brown (1987) keeps only the minimal hitting sets at each state. The number and size of the minimal hitting sets may increase or decrease as more evidence is found. Let Θ be the frame of discernment or the set of all possible broken components¹⁷. The space in which the diagnosis algorithm has to search is the power set of Θ . As shown in Figure 17.13, when the device is working the empty set Φ is the only minimal candidate. This is interpreted as saying that having no broken component is consistent with the observations. Here onwards, to facilitate drawing in figures, we shorten the names of the five components of Figure 17.12 to M_1 , M_2 , M_3 , A_1 , and A_2 .

The following two tasks are the core of the diagnosis algorithm.

1. Conflict Recognition Given an observation OBS , to identify the set of all minimal conflicts for the device. Further, when more observations arrive, to efficiently augment the set of minimal conflicts.

2. Candidate Generation Given a set of minimal conflicts, to identify the candidates for diagnosis. The candidates are the minimal hitting sets corresponding to the set of conflicts. Further, as more conflicts are generated, to modify the candidates in an efficient manner.

Candidate Generation

We first look at candidate generation. Initially, the set of minimal candidates contains only Φ . At any point, given a set of minimal candidates and a new minimal conflict, the following procedure is adopted.

- Any minimal candidate that does not explain the new conflict is replaced by one or more supersets of the candidate. Each superset is composed by adding one element of the new conflict to the failed candidate.
- If any candidate in the new set of candidates is subsumed by another candidate, then it is removed.

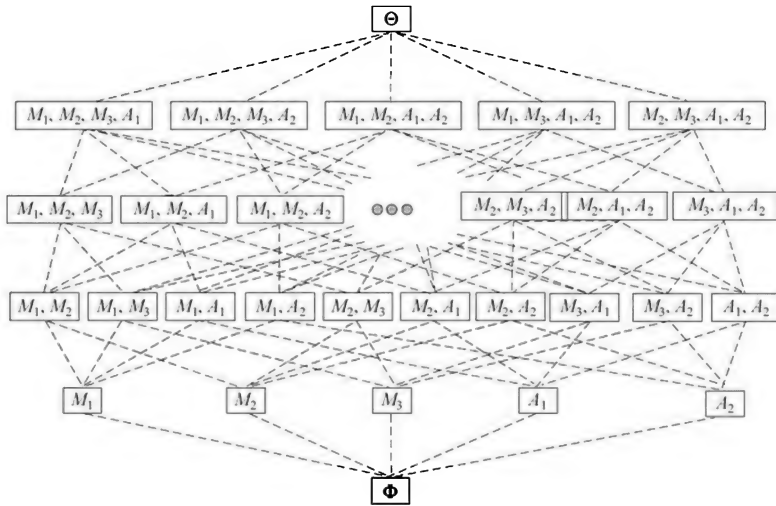


Figure 17.13 The candidate space in diagnosis. Any subset of the five components may be a candidate. The minimal candidate when the device is working is the empty set.

Let us first look at how GDE deals with two conflict sets $\langle A_1, M_1, M_2 \rangle$ and $\langle A_1, A_2, M_1, M_3 \rangle$ given to it. When it gets the first conflict, there is only one minimal candidate, the empty set, and it does not explain the conflict. Hence it is removed, and replaced by three candidates— $\{M_1\}$, $\{M_2\}$, and $\{A_1\}$ —which are singleton candidates. The situation is shown in Figure 17.13.

The interpretation of this set of candidates is any of them or any of their ancestors could represent the broken set and that would be consistent with the observation (in this case the set of input and output values). This implies that the supremum Θ is always a candidate for being broken, but that only tells us that the entire set of candidates is broken, or that the device is broken. The reason why we only consider the minimal candidates as diagnosis is that if that candidate is the diagnosis, then the components in that are necessarily broken.

Observe that $\{M_3\}$, $\{A_2\}$ and $\{M_3, A_2\}$ are neither candidates nor are they the ancestors of any candidate. They are in the clear, and not suspects any more. This means that by themselves, they cannot explain all the conflict sets. The candidate $\{\Phi\}$ is also out of contention. It cannot be the case that no component is broken, because that would be inconsistent with the observations (or the conflict sets).

Next, the second conflict $\langle A_1, A_2, M_1, M_3 \rangle$ has to be processed. Of the three candidates in Figure 17.14, $\{A_1\}$ and $\{M_1\}$ can explain the conflict since each has one element in the conflict. The remaining candidate $\{M_2\}$ does not, and has to be discarded. The following extensions have to be considered instead— $\{M_2, A_1\}$, $\{M_2, A_2\}$, $\{M_1, M_2\}$ and $\{M_2, M_3\}$. That is, if M_2 is broken then one of A_1 , A_2 , M_1 , and M_3 must also be broken. Of these candidates, $\{M_2, A_1\}$ is subsumed by (is a

superset of) $\{A_1\}$, and $\{M_1, M_2\}$ is subsumed by $\{M_1\}$. They are not minimal and have to be discarded. The remaining two are added to the set of candidates which now constitutes of $\{A_1\}$, $\{M_1\}$, $\{M_2, A_2\}$ and $\{M_2, M_3\}$, as shown in Figure 17.15. The reader should verify that these are the four minimal hitting sets for the two conflicts sets we started with. The precise algorithm for updating the set of candidates given a new conflict set is left as an exercise.

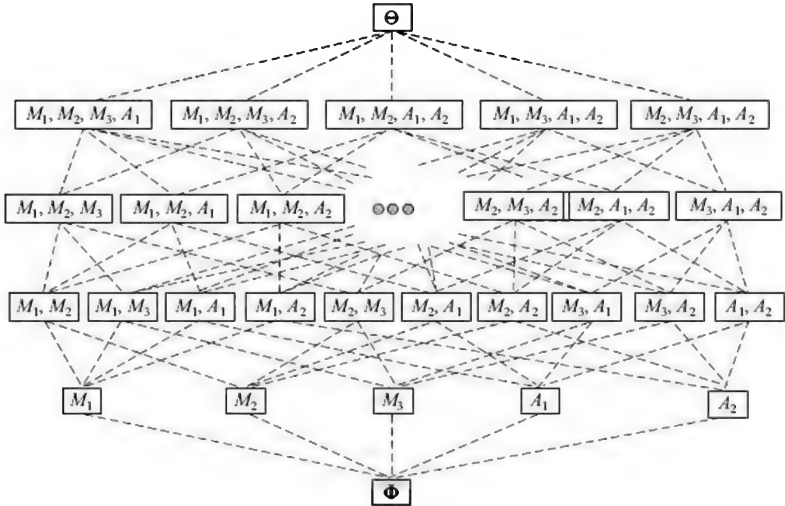


Figure 17.14 After the first conflict $\langle M_1, M_2, A_1 \rangle$ is processed, the minimal candidates are $\{M_1\}$, $\{M_2\}$, and $\{A_1\}$, while $\{F\}$, $\{M_3\}$, $\{A_2\}$ and $\{M_3, A_2\}$ are out of contention.

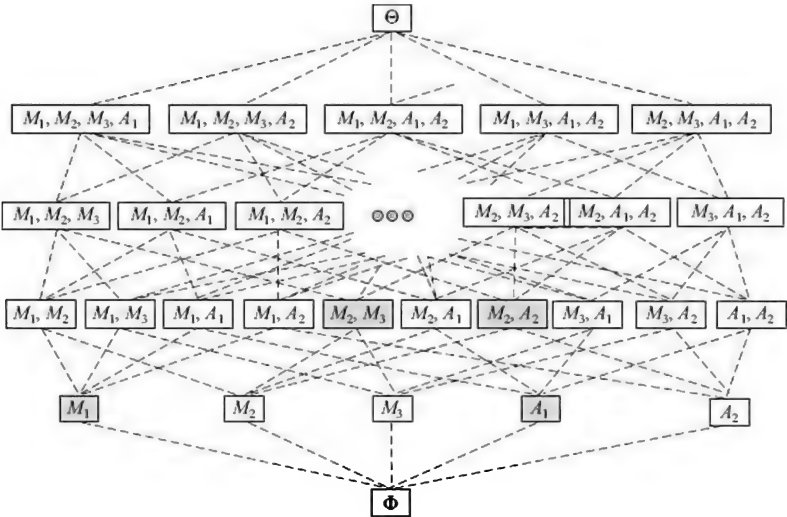


Figure 17.15 After the second conflict $\langle A_1, A_2, M_1, M_3 \rangle$ is processed, the minimal candidates are shown in the shaded rectangles, while the exonerated ones have thick borders.

When we begin with a correctly functioning device, the only (minimal) candidate for diagnosis is Φ . Once a discrepancy is found between the predicted and observed behaviour, and a conflict set generated, a new set of candidates is computed, and Φ , along with possibly a few others is no longer a candidate. Once a candidate is out of contention, it remains out of contention as more data in the form of conflicts is received. The set of exonerated components (or assumptions in the logic model) grows monotonically bottom up in the lattice. One can think of the process of diagnosis as the process of exonerating candidates, till only one (minimal) candidate remains.

Conflict Generation

The candidate generation algorithm takes conflicts as inputs. A conflict (or a conflict set) is derived from an observation. If the observation has a discrepancy contained in it, then it will lead to a nonempty, conflict set. A conflict set is derived from two ways at arriving at the value of a variable in a system. In the example we have been using, the variables are values at the terminals of some component.

The first conflict $\langle A_1, M_1, M_2 \rangle$ is arrived at by observing the discrepancy in the two values at F_1 in Figure 17.12. One value $F_1 = 12$, is based on the assumptions $\neg Ab(M_1)$, $\neg Ab(M_2)$ and $\neg Ab(A_1)$, which state that the corresponding components are working correctly and not broken. The other value $F_1 = 10$ is an observation, and is based on no assumption. The cumulative assumptions from the two ways of arriving at the value are inconsistent together. That is $\neg Ab(M_1)$, $\neg Ab(M_2)$ and $\neg Ab(A_1)$ cannot be true at the same time. We represent this fact by the corresponding set $\langle \neg Ab(M_1), \neg Ab(M_2), \neg Ab(A_1) \rangle$, which in the abbreviated form that we have been using is $\langle M_1, M_2, A_1 \rangle$.

The second conflict $\langle A_1, A_2, M_1, M_3 \rangle$ is arrived at two ways of generating the value of F_2 . The first is the observation $F_2 = 12$, which has no assumptions. The second $F_2 = 10$ is derived as follows.

	$F_2 = D_2 + E_2$	Assumption: $\neg Ab(A_2)$
where	$E_2 = A_3 + B_3$	Assumption: $\neg Ab(M_3)$
and	$D_2 = E_1$	because $D_2 = C_2$ and $C_2 = E_1$
and	$E_1 = F_1 - D_1$	Assumption: $\neg Ab(A_1)$
and	$F_1 = 10$	Observation
and	$D_1 = A_1 \times B_1$	Assumption: $\neg Ab(M_1)$
and	$A_1 = 3$ and $B_1 = 2$ and $A_3 = 3$ and $B_3 = 2$	

The statement $D_2 = E_1$ is not based on any assumption because our model assumes that only components can fail and not conduits. The cumulative set of assumptions in the reasoning above gives us the conflict $\langle A_1, A_2, M_1, M_3 \rangle$. Observe that this conflict could also have been

derived by comparing two ways of arriving at a different value of C_2 . See Exercise 12. At the same time, we treated the value at F_2 differently from the value at F_1 . This was because a symmetric treatment, the observed value and the one derived from M_2 , M_3 and A_2 are the same, and would not lead to a conflict set.

The manner in which the second conflict has been derived gives us a clue of how to go about finding (minimal) conflicts. The key is that one must find two ways of deriving different values of some variable, and enumerate the components that played a role in the two derivations. The set of components, or more precisely the assumptions that they are not broken, forms a conflict set. In the process of exploring different variables, one will have to discard (a) conflicts that are replicated, and (b) conflicts that are subsumed (are supersets of) other conflicts.

In their 1987 paper describing the *GDE*, de Kleer and Brown introduce the notion of an *environment* which stands for a set of assumptions. If the environment leads to a contradiction (a discrepancy) then it becomes a conflict set.

Since one is interested only in the minimal conflicts, one can explore the environments starting with the smallest. At any given point if an environment becomes inconsistent then it becomes a conflict.

Inconsistency can be checked using a proof procedure (for example, the resolution refutation method augmented to recognize the contradiction in clauses like $6 + 6 = 10$). If an environment becomes a conflict then its ancestors in the lattice too need not be investigated. Otherwise, it is refined into several environments by adding a new assumption. As a corollary, if some descendent (generalization) of an environment is a conflict, then it need not be investigated.

GDE employs an inference procedure in which the inferred sentences, or *beliefs*, are supported by an environment in which they are inferred, and the support is noted explicitly as a *justification*. Let $P(OBS, ENV)$ be the set of inferences made by the system, given the observations *OBS* and the environment *ENV*.

One of the ways of generating new data is to keep input values constant and make some additional measurements at some internal terminal in the device¹⁸. When one does that then the inferences made by the system before the measurement *M* still hold, and a few additional inferences may be made. That is,

$$P(OBS, ENV) \subseteq P(OBS \cup \{M\}, ENV)$$

This implies that as we make new measurements, we need to do the inferences only incrementally, as long as we cache the inferences made earlier. Further, since the inferences are made for specific environments, if we store the justification for each inference, then when an inconsistency is observed, the environments responsible for it can be identified easily.

An environment is a set of assumptions. If the environment *ENV* turns out to be consistent (with the observations and system definition), then

the system investigates a set of new environments of the form $ENV \cup \{A\}$, where A is a new assumption. The inferences made with the smaller environment are still valid and,

$$P(OBS, ENV) \subseteq P(OBS, ENV \cup \{A\})$$

Thus, while investigating the extended environments, the cached inferences can be augmented with any new ones. Further, since environments are explored smallest first, when a new environment comes up, all its “subsets” already have the inferences made and cached. What is needed is a mechanism that stores the inferences and alerts the system not to repeat the same inferences again. Such a mechanism is called a *Truth Maintenance System* (TMS) and we will study it briefly in the next section. In fact, the *GDE* employs a version of *TMS* known as the *ATMS* (*Assumption based TMS*) that allows it to explicitly keep track of assumptions (environments).

GDE does not explore all possible environments, but only those that are suggested by the structure of the device which indicates which components are connected. The relations between variables across components are expressed in the form of rules or constraints. Each such rule or constraint identifies a component that can be part of an environment when the rule is applied or the constraint propagated across it. Further, propagation happens only through those components that are connected, and only those can extend a given environment. In this way, the connectivity guides the extensions in the environments.

Measurements

There are two ways that conflicts are generated. One, when a discrepancy occurs between an observed value and a predicted value. Then tracing the justifications of the derived values an environment is assembled, which may lead to a conflict. *GDE* caches the reasoning it does, while generating predictions as justifications in the *ATMS*. This means that while exploring for conflicts, it can exploit the stored justifications. For each discrepancy, the system searches for different environments (assumptions) that are consistent with the discrepancy. In the example we have been following, first tracing back the derived values to the inputs leads to $\langle M_1, M_2, A_1 \rangle$. Next, instead of tracing E_1 via its source C_2 , the search tries to trace (propagate) it via D_2 . This leads to dropping M_2 and adding A_2 and subsequently M_3 to the environment.

Second, when one has exhausted the (minimal) conflicts that can be generated by a set of observations, then additional observations or measurements need to be made. In our example, the measurement can be made at one of three internal points. The first is C_1 , or equivalently D_1 , since conduits are assumed to be free of faults. The second is E_1 , or equivalently C_2 or D_2 . The third is at E_2 or equivalently C_3 .

Let us say we make a measurement at C_1 . Consider three different values that the measurement could yield.

- $C_1 = 4$. This will lead to a conflict $\langle M_1 \rangle$. This would in turn imply $Ab(M_1)$ and the minimal candidate would be $\{M_1\}$. The value $F_1 = 10$ would be consistent with $C_1 = 4$.
- $C_1 = 6$. This would exonerate M_1 and would lead to a conflict $\langle A_1, M_2 \rangle$, or $\langle A_1, A_2, M_3 \rangle$. The reader should verify that the minimal candidates now are $\{A_1\}$, $\{M_2, M_3\}$ and $\{M_2, A_2\}$.
- $C_1 = 7$ or any value not equal to 4 or 6. This will also lead to a conflict $\langle M_1 \rangle$ and imply $Ab(M_1)$. The value $F_1 = 10$ however would be inconsistent with $C_1 = 8$ and would lead to the conflicts $\langle A_1, M_2 \rangle$, or $\langle A_1, A_2, M_3 \rangle$. This would be a case of more than one fault in the system. The minimal candidates would be $\{M_1, A_1\}$, $\{M_1, M_2\}$ and $\{M_1, M_2, A_2\}$.

GDE keeps track of inferences that enable it to identify possible points for measurement. The details of how the ATMS represents data are described in Section 17.4. In our example, it keeps the following dependencies in the form [Value, Supporting-environment(s)].

$[C_1=4, \{\{M_2, A_1\}, \{A_1, A_2, M_3\}\}]$
 $[C_1=6, \{M_1\}]$

$[C_2=4, \{\{A_1, M_1\}\}]$
 $[C_2=6, \{\{M_2\}, \{A_2, M_3\}\}]$

$[C_3=8, \{\{A_1, A_2, M_1\}\}]$
 $[C_3=6, \{\{M_3\}, \{A_2, M_2\}\}]$

Every variable that receives different values is a candidate for measurement. The question is which variable to measure next. The answer will depend upon many factors. Chief amongst them is the cost of making the measurement. If making a measurement is cheap, then one could go ahead and make all possible measurements. However, this is not often the case, and one may have to choose measurements carefully. The best measurement would be the one that eliminates half the candidates for diagnosis. If this could be done then one would arrive at a diagnosis with the smallest number of measurements on the average.

The decision of where to take the measurement will also depend upon knowledge of which components are more likely to fail. If that information is available then terminals which have the most likely to fail component in their environment can be given preference. If the *a priori* probabilities are known then a minimum entropy approach can be used to select a measurement point (de Kleer and Brown, 1987). The details of this approach are beyond the scope of this text. However, we present a simpler approach presented by de Kleer (1990) in a subsequent paper. The approach is based on the following assumptions.

1. All components fail independently and with equal probability.
2. Each component fails with an extremely small probability.

3. We are interested in discriminating only between the smallest cardinality diagnosis candidates that remain in the fray.

Given the assumption that components fail with an extremely small probability, the diagnosis which says that the smallest number of elements has failed is the most credible. This can be seen to follow from *Occam's razor*¹⁹ or the *principle of parsimony* that says that the simplest explanation is most likely to be the correct one²⁰.

The approach presented by de Kleer is as follows. Let us say that in the set of minimal candidates for diagnosis the smallest cardinality of a diagnosis is Q . Let X be a possible measurement point with possible values $\{v_1 \dots v_k\}$. Then, the score $S(X)$ for making a measurement of X is,

$$S(X) = \sum_{1 \leq i \leq k} C_i \ln(C_i)$$

where C_i is the number of diagnosis of cardinality Q which predict the value v_i , and \ln is the natural logarithm. The measurement point with the lowest score is the best candidate.

In our example, given that $F_1 = 10$ and $F_2 = 12$, there two lowest cardinality diagnosis candidates, $\{M1\}$ and $\{A_1\}$, and $Q = 1$. The set of values and the corresponding singleton diagnosis that predict them are,

$$\begin{aligned} C1 &= 4 \leftarrow \{M_1\} \\ C1 &= 6 \leftarrow \{A_1\} \\ C_2 &= 6 \leftarrow \{A_1\}, \{M_1\} \\ C_3 &= 6 \leftarrow \{A_1\}, \{M_1\} \end{aligned}$$

The last two sentences should be read as $\{M_1\}$ predicts $C_2 = 6$, as does $\{A_1\}$. Therefore,

$$\begin{aligned} S(C1) &= 1 \times \ln(1) + 1 \times \ln(1) = 0 \\ S(C_2) &= 2 \times \ln(2) = 1.4 \\ S(C_3) &= 2 \times \ln(2) = 1.4 \end{aligned}$$

indicating that measuring at C_1 is the best choice. The interested reader can find the derivation of the above expression in de Kleer's paper.

17.3.4 Fault Models

The consistency based diagnosis approach we have seen so far works with models in which component behaviour is described only for the working case. The abnormality predicate $Ab(C)$ can only distinguish between working and nonworking components. As far as it knows, either component C works or it is broken. If it works then its behaviour must conform to the constraints or relation associated with it. When the component is broken, it is assumed that nothing can be said about its behaviour.

In practice, one can say something about the behaviour of broken components. And one can often identify different ways of a component malfunctioning, each with a characteristic behaviour pattern. For example, the pressure regulator valve discussed earlier could get stuck in the OPEN position, in which pressure drop would be proportional to the flow of the fluid. Or it could get stuck in the CLOSED position, in which case the flow would be zero.

As shown by Peter Struss and Oskar Dressler (1989), an understanding of the behaviour of a broken component, which they term *physical negation*, can lead to pruning of impossible candidates that GDE is compelled to keep amongst the contenders.

We look at the example discussed in their paper. The circuit shown in Figure 17.16 is made up of three kinds of components. There are three bulbs, six wires (which are now treated as components and not conduits), and one battery. The ten components are connected as shown in the figure. We assume there are no conduits, and each component has two terminals which may directly be connected to the terminals of other components. The situation is that bulb B_3 is lit, while B_1 and B_2 are off. The task is to find out what is broken.

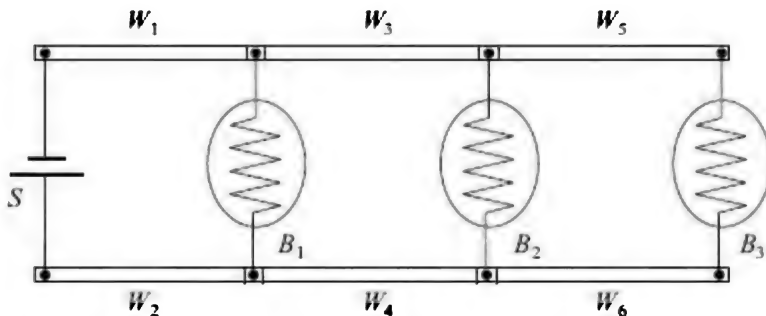


Figure 17.16 Three bulbs connected to a battery in parallel, with six wire components.

To anyone with a basic knowledge of electrical circuits, it is immediately clear that both the bulbs B_1 and B_2 must be broken. The reason is that wires and the battery must all be working because the bulb B_3 is lit. However, working only with descriptions of working component models, GDE is unable to pinpoint this diagnosis, though it does identify it amongst many spurious ones. Struss and Dressler show that even with assuming that there is only one fault mode, a description of the component behaviour can help. In the general case, there could be many different ways that a component could fail in, and the task would be to identify what behaviour mode the component is.

Other papers that have adopted similar approaches are (de Kleer and Williams, 1989), (Dvorak and Kuipers, 1989), (Hamscher, 1991), and (de Kleer and Brown, 1992). A comprehensive collection of papers is the Readings in Model Based Diagnosis edited by (Hamscher et al., 1992).

We begin by examining how GDE diagnoses the problem.

Let us model the components as constraints. We assume two modes for each component. One $OK(C)$ that corresponds to $\neg Ab(C)$ in our earlier notation, and the other $Broken(C)$ that corresponds to $Ab(C)$. This notation allows us to identify the mode of fault, and we could have different modes $Broken_1(C)$, $Broken_2(C)$, ..., $Broken_F(C)$, if there were F different possible fault modes. With each fault mode, we would be expected to describe at least some aspects of the behaviour.

GDE describes the behaviour only for the OK mode. Let us build a qualitative model in which the voltage values are from the quantity space $[-, 0, +]$. When the battery S is working then the voltages at its two terminals are $Voltage(T_+) = +$ and $Voltage(T_-) = -$. For the sake of brevity, we will write these as $T_+ = +$ and $T_- = -$.

Table 17.8The working bulb model

<i>Mode</i>	<i>L</i>	<i>R</i>	<i>Bulb</i>
OK	+	-	On
OK	-	+	On
OK	+	+	Off
OK	-	-	Off
OK	0	0	Off
OK	0	+	Off
OK	0	-	Off
OK	+	0	Off
OK	-	0	Off

The bulb is modelled by the relation depicted in Table 17.8. We have included the mode OK in the relation. Later when we extend the behaviour description to the fault mode, we can extend the table for the mode $Broken$. The other columns are L and R for the two terminals the values of which are voltages, and $Bulb$ for the state of the bulb which can take two values, On or Off . The value 0 for Voltage means that the battery voltage is not being transmitted to it. The astute reader would have observed that this does not conform to the no-function-in-structure principle. In particular, this model would not work for bulbs connected in series, since that would need a denser quantity space that would allow for a voltage drop across each bulb.

The corresponding working model for the wire says that when the wire is on the OK mode, then the voltage at both its ends is the same.

Table 17.9 The working wire model

Mode	L	R	Wire
OK	+	+	Connected
OK	-	-	Connected
OK	0	0	Connected

Each of the bulbs B_i has the associated terminals TB_i and BB_i referring to the terminal on top of B_i and bottom of B_i respectively in Figure 17.16. Likewise, the two terminals of each wire W_i are named LW_i and RW_i for the left end and the right end respectively. Then our system description contains statements like $RW_1=TB_1$ to assert the fact that the top terminal of bulb B_1 is connected to the right end of wire W_1 . In a similar way, $RW_1=LW_3$ says that the right end of the wire W_1 is connected to the left end of the wire W_3 . These statements are really short forms for the statements $Voltage(RW_1)=Voltage(TB_1)$ and $Voltage(RW_1)=Voltage(LW_3)$. The reader is encouraged to write the complete system description SD for the above circuit.

The observations are $\{B_1=Off, B_2=Off, B_3=On\}$.

GDE constructs the following minimal conflicts.

- $\{S, W_1, W_2, B_1\}$ since the bulb B_1 is off, while it is predicted to be on.
- $\{S, W_1, W_2, W_3, W_4, B_2\}$ since the bulb B_2 is off, while it is predicted to be on.
- $\{B_3, W_5, W_6, B_2\}$ since voltage values around this loop are inconsistent.

Because B_3 is on it is expected that

$$TB_3 = RW_5 = TB_2 = LW_5 = \dots = (T_- = -)$$

$$BB_3 = RW_6 = BB_2 = LW_6 = \dots = (T_+ = +)$$

But $TB_2 = -$ and $BB_2 = +$ is not possible because the bulb B_2 is off.

- $\{B_3, W_3, W_4, W_5, W_6, B_1\}$ in a similar manner because B_3 is on and B_1 is off.

As reported in (Struss and Dressler, 1989), GDE finds many minimal candidates including the following with the lowest cardinality 2,

$$\{S, B_3\}, \{S, W_5\}, \{S, B_6\}, \{W_1, B_3\}, \{W_1, W_5\}, \{W_1, W_6\}, \{W_3, B_3\}, \{W_2, W_5\}, \{W_2, W_6\}, \{B_1, B_2\}$$

The first one $\{S, B_3\}$ illustrates the consequence of ignorance about the physical manner in which components can be broken, that is, not having fault models. One can interpret the candidate in the following way, keeping in mind that GDE only has descriptions of working behaviour, and when a component is broken, any behaviour is possible. The fact that the battery S is broken explains the observations that bulbs B_1 and B_2 are off. The observation that B_3 is lit is "explained" by the fact that it is broken (if it were OK then it would have been off too!). In a similar

manner, the fact that W_2 is broken (in the candidate $\{W_2, W_5\}$) explains the fact that B_1 and B_2 are off, while the wire W_5 being broken somehow explains bulb B_3 being lit.

From a purely logical standpoint, the above reasoning is consistent, because GDE uses only working component models. But from an understanding of the physical systems, it is apparent the bulbs cannot light up by themselves, or because a wire connecting them is broken.

The behaviour model for the bulb is extended in Table 17.10 to one fault mode Broken below. The key feature in the table is that if the bulb is Broken then in all cases, the bulb stays in the *Off* state.

Table 17.10 The broken bulb model

<i>Mode</i>	<i>L</i>	<i>R</i>	<i>Bulb</i>
Broken	+	–	Off
Broken	–	+	Off
Broken	+	+	Off
Broken	–	–	Off
Broken	0	0	Off
Broken	0	+	Off
Broken	0	–	Off
Broken	+	0	Off

Struss and Dressler extend the GDE to GDE+ by incorporating fault models in the following manner. Assuming that we represent $\neg Broken_i(C)$ with $OK_i(C)$, the following statements are added for each fault model.

$$\forall i \leq F (OK(C) \supset OK_i(C))$$

and conversely,

$$OK_1(C) \wedge OK_2(C) \wedge \dots \wedge OK_F(C) \supset OK(C)$$

This is equivalent to saying that,

$$Broken(C) \supset Broken_1(C) \vee Broken_2(C) \vee \dots \vee Broken_F(C)$$

and conversely,

$$\forall i \leq F (Broken_i(C) \supset Broken(C))$$

In our example, $F = 1$ and there is only one fault mode (which we have called Broken). When GDE+ looks at the task, it too arrives at the same set of conflicts. However, the set of candidate diagnoses it generates, does not have the spurious diagnoses. Consider the candidate $\{S, B_3\}$. This is not generated by GDE+ because $Broken(B_3)$ is

not consistent with the observation $B_3 = \text{On}$, because no row in the fault model allows $B_3 = \text{On}$. For the same reason, all candidates containing B_3 are rejected.

That leaves the candidates with two broken wire segments. Table 17.11 below defines the broken wire model. While the label “Disconnected” does not play a role, the voltage values for the two ends do matter.

Table 17.11 The broken wire model

<i>Mode</i>	<i>L</i>	<i>R</i>	<i>Wire</i>
Broken	+	–	Disconnected
Broken	–	+	Disconnected
Broken	+	0	Disconnected
Broken	0	+	Disconnected
Broken	–	0	Disconnected
Broken	0	–	Disconnected

Consider the candidate $\{W_2, W_5\}$ found by GDE. Given that $LW_2=+$ (since the battery is working) and the assumption that *Broken* (W_2), there is no way that $RW_2=+$ and consequently no way that $BB_3=+$, which in this example is a necessary condition for bulb B_3 to be lit. By a similar argument, none of the wires can be broken if the B_3 is to be on, and all candidates that have a wire are eliminated.

What remains is the candidate $\{B_1, B_2\}$.

Struss and Dressler illustrate the use of multiple fault modes by the following example. In an adaptation shown in Figure 17.17, a water filtering system receives water through an input pipe *P-IN* from a source²¹ *D* that is unbounded, or at least of a few orders of magnitude greater capacity. Filtered water flows out from *P-OUT*. If there is a leak or an overflow, the water falls into a catchment area *O* where it is sensed and an alarm raised. The observations are that *Output(P-OUT)* = 0 and *Alarm(O)*=Off. What has gone wrong?

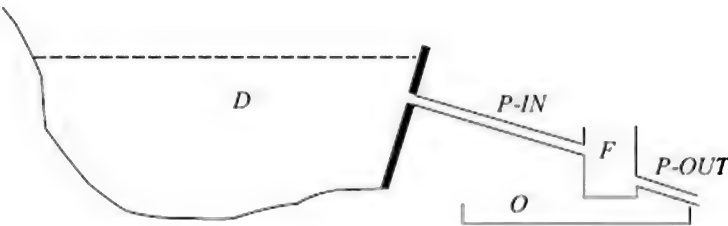


Figure 17.17 A water filtering system *F* gets input from a nearby dam *D* and sends the filtered water through *P-OUT*. An overflow catchment area *O* senses water and generates an alarm.

The working models of $P-IN$, F , and $P-OUT$ entail that $Output(P-OUT)=+$. So, there is a discrepancy between the observation and the prediction. A new case for GDE to solve.

GDE generates the only conflict $\langle P-IN, F, P-OUT \rangle$, resulting in the three candidates $\{P-IN\}$, $\{F\}$, and $\{P-OUT\}$. This only says that one of the three components involved in supplying water is broken.

Let us hand the case over to GDE+. The following fault models describe some of the ways in which components may be broken. The names chosen are self explanatory. The details of their effects have been left out.

- $Hole(F) \equiv Broken(F)$. The behaviour is that water leaks out of the tank in falls into the catchment area O , where it should be detected giving us $Alarm(O)=On$.
- $Perforated(Pipe) \equiv Broken_1(Pipe)$. Likewise, one of the pipes may have a hole, letting the water fall into O and triggering the alarm.
- $Blocked(Pipe) \equiv Broken_2(Pipe)$. In this mode of failure, the pipe gets blocked and no water flows through it.

The given observations are $\{Output(P-OUT) = 0, Alarm(O)=Off\}$. GDE+ constructs the following additional conflicts $\langle P-IN, O \rangle$. This says that at least one of $P-IN$ and (the alarm in) O must be broken. We look at how this is arrived at in a little more detail in the following section.

Given the two conflicts, there are three candidates for diagnosis,

- $\{P-IN\}$ — The input pipe is blocked.
- $\{F, O\}$ — The filter tank F has a hole *and* the alarm in O is not working.
- $\{P-OUT, O\}$ — The output pipe is perforated *and* the alarm in O is not working²².

The important thing is that the GDE+ can discriminate between different fault modes and come up with more information about what is the nature of the fault.

Fault models therefore have their advantages. The main disadvantage is that the space of candidates explodes. When there were only two modes OK and $\neg OK$, the candidate space was of size 2^N when there are N components. With each component admitting F faulty modes, this space grows to $(F+1)^N$. Another perceived drawback is that all fault modes must be enumerated for completeness. However, this has been countered by saying that an uninformative fault mode, equivalent to $\neg OK$, can be added as well.

17.4 Assumption Based Reasoning and Truth Maintenance

A problem solver that does some reasoning in a domain may have to make assumptions. An assumption is a belief. It may not necessarily be true. Based on the assumption, the reasoning system may infer other beliefs, and still others from those beliefs. It is possible that at some

point, the set of beliefs held by the problem solver becomes inconsistent. It then becomes necessary for the problem solver to retract some assumption.

A Truth Maintenance System (TMS) (Doyle, 1979) is a program that does the book keeping and consistency management for a problem solver²³ (Figure 17.18). For every sentence the problem solver accepts or generates, the TMS maintains a node in its network. The sentence, referred to as the datum, is a proposition including statements like $C_1=3$, Culprit(butler), and Man(socrates) ... Mortal(socrates). When the problem solver, which could be a theorem prover, a constraint solver, or a custom made reasoning engine, generates a new piece of datum, it informs the TMS about the new datum along with a *justification* that names the data items or antecedents that resulted in the derivation as well as the method or rule that was used. A justification is of the form,

<consequent, informant, antecedent₁, ..., antecedent_k>

This representation is reminiscent of the way we write proofs, stating the consequent followed by how it was derived. The TMS itself treats each piece of data simply as a node in its network of justifications, and is not concerned with its semantics. It is only concerned with keeping track of justifications and maintaining consistency. If a special node standing for a contradiction receives justification, it informs the problem solver and triggers a *Dependency Directed Backtracking* (see also Chapters 6 and 9) routine to identify the source of inconsistency that the problem solver can jump back to.

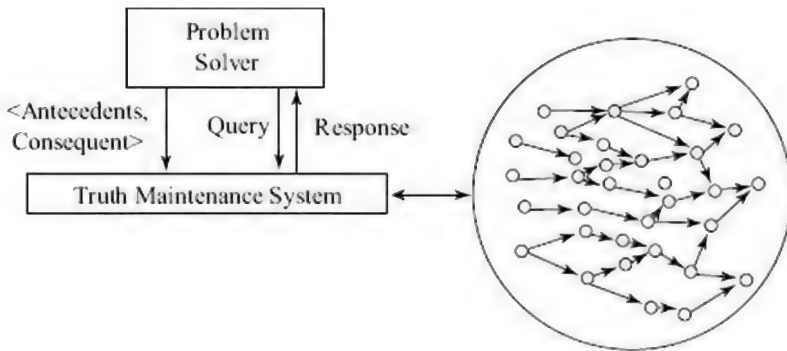


Figure 17.18 A Truth Maintenance System keeps track of justifications of all beliefs held by the Problem Solver and keeps a watch on the consistency of the beliefs.

Consider the following (relevant) excerpts from a fictitious set of facts.

1. Nira spoke either to Prashant or to Barkha.
2. Barkha spoke either to Kapil or to Azad.
3. Vir spoke either to Vinod or to Nira.
4. If Nira spoke to Prashant, the culprit is Nitish.
5. If Nira spoke to Barkha, the culprit is in Chennai.

6. If Barkha spoke to Kapil, the culprit is Singh.
7. If both Singh and Nitish are the culprits, the document is in Patna.
8. If Vir spoke to Vinod, the culprit is Surinder.
9. If Vir spoke to Nira then the story got published.
10. Smiley said that the document is in Delhi.
11. If Smiley said the document is in Delhi then the document is in Delhi.
12. The document cannot be both in Patna and Delhi at the same time.

Given the either-or statements 1 to 3, let us say a reasoning system makes *some* assumptions about one of the constituents in these statements and derives the consequences. The reasoning, let us say, happens as in the following sequence. We have not stated the rules of inference used here, but the reader is encouraged to formalize the derivation of the contradiction.

13. Nira spoke to Prashant; Assumption
14. Barkha spoke to Kapil; Assumption
15. Vir spoke to Vinod; Assumption
16. Nitish is the Culprit; Logic rule ; 4, 13
17. Singh is the Culprit; Logic rule ; 6, 14
18. The document is in Patna; Logic rule ; 7, 16, 17
19. Surinder is the Culprit; Logic rule ; 8, 15
20. The document is in Delhi; Logic rule ; 19, 11
21. Contradiction; Logic rule ; 12, 18, 20

At this point, a contradiction has been signalled, and the reasoner needs to backtrack. The dependency diagram based on the justifications looks as in Figure 17.19. In the figure, sentences in ovals represent beliefs. Sentences in ovals with open arrows leading into them are premises and need no justification. A special node containing \perp stands for a contradiction. Assumptions are depicted in rectangular boxes. Dashed arrows represent any other inference chains that could have been constructed, but are not depicted here.

The *TMS* provides the following services to the problem solver.

- It caches the derivations made by the problem solver. This means that if the same reasoning step is to be made again, the *TMS* can help avoid duplicated computations.
- When a contradiction is signalled, it facilitates identification of possible causes and dependency directed backtracking.
- Once a node is introduced, it stays in the network. It may be marked *IN* signifying that there is positive belief for the datum, or *OUT* signifying negative belief. Nodes may change status multiple times during problem solving.
- A node is *IN* if at least one justification is supported by nodes that are *IN*. Else it is *OUT*. At any time, the *TMS* can inform the problem solver whether a node is being believed or not. That is, whether it is *IN* or *OUT*.
- Once an assumption is retracted, it is marked *OUT*. The network of justifications is used to propagate this change to succeeding nodes.

- A set of assumptions that leads to a contradiction is marked as a *nogood* and never repeated again²⁴.

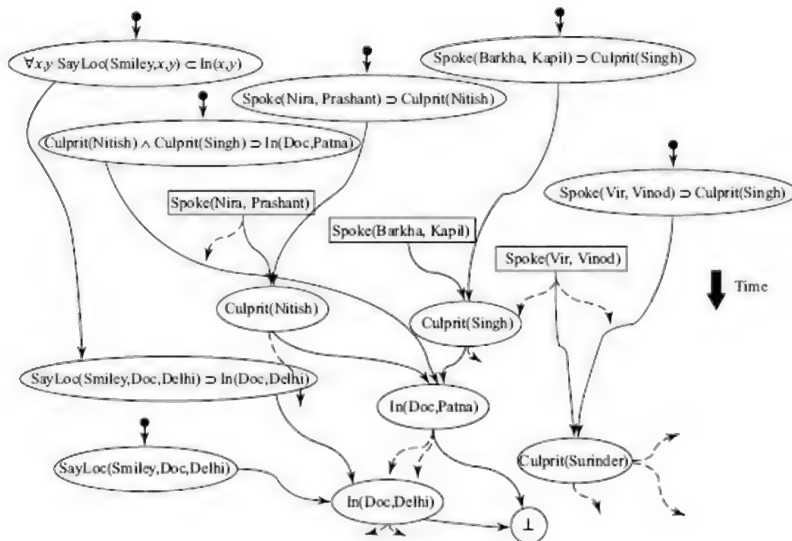


Figure 17.19 The justification links to the contradiction node enable dependency directed backtracking. Assumptions are shown in rectangular boxes.

The reasoning system described above has made the assumptions in the following order,

1. Nira spoke to Prashant ; Assumption
2. Barkha spoke to Kapil ; Assumption
3. Vir spoke to Vinod ; Assumption

Chronological backtracking would retract the last made assumption first. However, in this case, the last assumption has no bearing on the inconsistency observed in the TMS. Instead, the algorithm traces back the justifications from the contradiction node and suggests that one of the other two assumptions needs to be retracted. Let us say that the second assumption *Spoke(Barkha, Kapil)* is retracted. The TMS does this by marking the node *OUT*. It then explores all the consequences of this change, looking for nodes that have no justification after the retraction. All such nodes are also marked *OUT*. The resulting network for our example is depicted in Figure 17.20. The nodes that are *OUT* are shown in shaded containers. Any nodes pointed to by the dashed arrows from these nodes would need to get their justifications checked too.

17.4.1 Assumption Based Truth Maintenance System

The TMS system, as constructed by Jon Doyle, keeps *one* consistent

network in its memory. Any assumption whose addition would lead to a contradiction and nodes justified only by it, are marked as out.

However, in many problems, the reasoner may have to work with different, and often contradictory, assumptions simultaneously. During the reasoning process, it may have to work with different sets of assumptions alternately.

The following simple example illustrates the case where different sets of assumptions can be made. Consider the following facts.

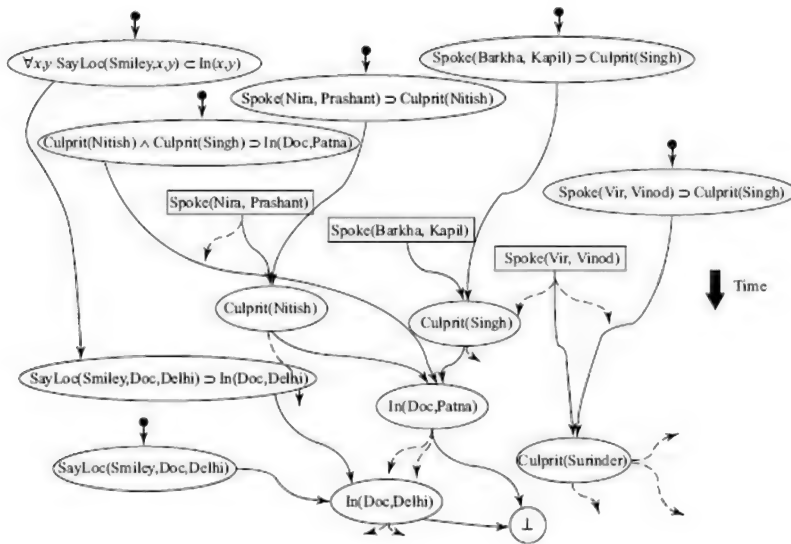


Figure 17.20 On discovering a contradiction, a relevant assumption is retracted and the change propagated in the network. The nodes marked OUT are shown as shaded nodes.

$$\begin{aligned}
 & \text{On}(a, \text{table}), \\
 & \text{On}(b, \text{table}), \\
 & \exists x(\text{On}(x, \text{table}) \wedge \text{Colour}(x, \text{blue})), \\
 & \exists x(\text{On}(x, \text{table}) \wedge \text{Colour}(x, \text{green})), \\
 & \forall x(\text{Colour}(x, \text{blue}) \oplus \text{Colour}(x, \text{green}))
 \end{aligned}$$

The following is then entailed,

$$\begin{aligned}
 & \mathbf{BColour}(a, \text{blue}) \models \mathbf{BColour}(b, \text{green}), \\
 & \mathbf{BColour}(a, \text{green}) \models \mathbf{BColour}(b, \text{blue}),
 \end{aligned}$$

As discussed in Section 17.1.4, this knowledge base has two extensions. One in which block *a* is blue, and the other in which block *a* is

green. The basic *TMS* can store only one of these extensions. If it is given the assumption *Colour(a, blue)*, it can store *Colour(b, green)* if the reasoner derives it. It will also store any other statements that follow. However, if an assumption *Colour(a, green)* is added, then a contradiction could be triggered, and the *TMS* would instruct the reasoner to withdraw one of the assumptions, and any consequents that follow.

The basic *TMS* is unable to handle the two contradictory assumptions together. If it needs to switch from one set of assumptions to another, it has to systematically retract the first set, invoking all the consistency algorithms alongside, and introduce the new set. In doing so, it may have to give up on stored inference patterns that are common between the different sets, and end up making those inferences repeatedly.

An *Assumption Based Truth Maintenance System* (*ATMS*) (de Kleer, 1986; 1986a; 1986b) can keep multiple sets of possibly contradictory assumptions in its memory. As a consequence, it can simultaneously deal with different sets of beliefs that may be mutually contradictory, and investigate their consequences²⁵. The *ATMS* refers to each set of assumptions as an *environment*, and the set of statements that are entailed by the environment and other facts as the *context*.

In Model Based Diagnosis (Section 17.3), we have seen a form of reasoning in which logical inferences have to be made in different contexts under different sets of assumptions or environments. Often the different contexts have many common inferences and the reasoning system could benefit from sharing the work done. Also, the reasoner may need to switch back and forth between contexts and would benefit tremendously if all the inferences made at any time are cached along with their justifications, so that one can trace the dependencies between different “facts”. This becomes particularly relevant in a nonmonotonic situation when one may have to make and retract assumptions.

Like the *TMS*, the *ATMS* too employs a special data structure called node for every datum the problem solver uses, including database entries, rules and procedures. For the problem solver, the node represents belief in the datum. However, in addition to marking the nodes as *IN* or *OUT* along with the justifications, the *ATMS* also keeps a *label* with it that contains the different environments in which the node holds.

Every datum n in the *ATMS* has a label $L = \{E_1, E_2, \dots, E_l\}$ which is a set of environments. The label is a succinct description of all the contexts in which the datum holds. Given an environment $E_i \in L$ and the set of justifications J provided by the problem solver to the *ATMS*, it follows that,

$$E_i, J \vdash n$$

or equivalently,

$$J \vdash E_i \rightarrow n$$

where $E_i \rightarrow n$ can be read as “ n follows from E_i ”. The implementation of

the ATMS is required to ensure that the labels satisfy the following properties.

Consistent A label L is consistent, if all its environments are consistent.

Sound A label L is sound, if n follows from each $E_i \in L$.

Complete A label L is complete, if for every consistent environment E such that $J \vdash E \rightarrow n$, there exists an $E_i \in L$ such that $E_i \subseteq E$.

Minimalist A label L is minimal, if there are no two $E_i, E_j \in L$ such that $E_i \subseteq E_j$.

In the discussion below, we focus on the labels and represent an ATMS node as

$$\text{node} = \langle \text{datum}, \{\text{environments}\} \rangle$$

in the style of (Forbus and de Kleer, 1992) which does not mention the full description

$$\text{node} = \langle \text{datum}, \{\text{environments}\}, \{\text{justifications}\} \rangle$$

as in (de Kleer, 1986), where a justification is the immediate set of predecessors of the node in the dependency network. In this representation, the datum is the common consequent of all justifications. The informant has not been depicted, and can be assumed to be implicit in the justification. The following kinds of nodes can be distinguished.

A node is a *premise*, if the inference engine has provided it with a justification with no antecedents. That is, a premise A may be represented as,

$$\text{premise } A = \langle A, \{\{\}\}, \{\{\}\} \rangle \text{ or in the simpler notation}$$

$$\text{premise } A = \langle A, \{\{\}\} \rangle$$

An assumption A is a node that has itself in its environment²⁶.

$$\text{assumption } A = \langle A, \{\{A\}\}, \{\{A\}\} \rangle \text{ or}$$

$$\text{assumption } A = \langle A, \{\{A\}\} \rangle$$

However, assumptions may also receive justifications, and if they do, then they are treated as other nodes and cannot be retracted as long as their justification is intact. For the sake of simplicity, we will consider only those assumptions that are unsupported by other nodes. Premises may also receive justifications, though they do not contribute to the belief system.

If there are N distinct assumptions that one can make, then the ATMS may have to contend with 2^N different environments. However, if a node holds in an environment E then it will also hold in an environment E' that is a superset of E . The ATMS therefore stores only the minimal environments. Figure 17.21, adapted from (Forbus and de Kleer, 1992) shows a set of assumptions that are *IN*, a set of derived nodes, their justifications and their minimal environments.

The small empty circles in the figure represent *AND* nodes. Node *O* has one justification with two nodes *K* and *L*. Node *K* itself can be derived in two ways, either from *A* or from *B*. Thus, node *O* would have two environments, $\{A, B\}$ and $\{B\}$. However since the former is a superset of the latter, it is discarded. Likewise, node *M* holds in two environments, $\{C, D\}$ and $\{E\}$. Both these contribute to node *Q* which consequently holds in the two environments $\{B, C, D\}$ and $\{B, E\}$. The reader should verify that these are minimal, and retracting any assumption would disrupt the corresponding environment.

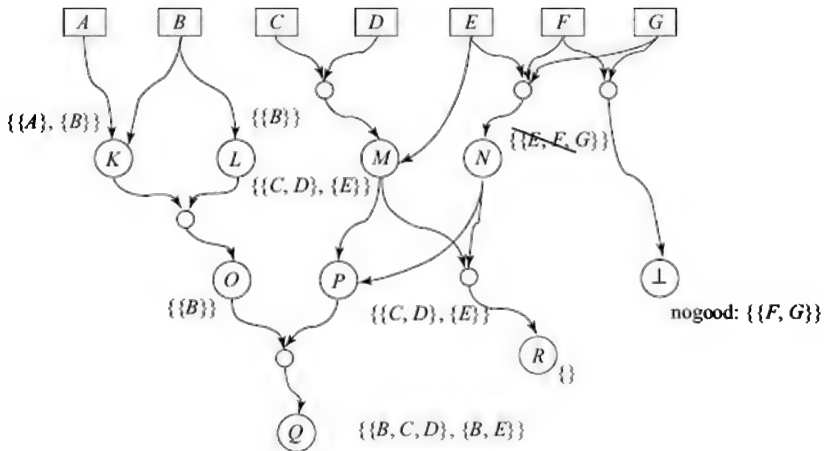


Figure 17.21 The environments for a set of statements. The small circles represent *AND* nodes.

The contradiction node is named \perp , and its environment is called a *nogood*.

$$\text{Contradiction} = \langle \perp, \{ \dots \}, \{ \dots \} \rangle$$

There may be several *nogoods* in the network and they represent the different combinations of assumptions that generate inconsistencies. In our example, the inference engine informs the *ATMS* that a contradiction resulted in the environment $\{F, G\}$. Consequently, the *ATMS* remembers the *nogood* $\{F, G\}$. Most implementations have a specific module for storing and querying *nogoods*. Now since $\{F, G\}$ is a *nogood*, so is $\{E, F, G\}$ being a superset of $\{F, G\}$. But $\{E, F, G\}$ leads to node *N*, which would have been an alternate way of deriving node *P*. But since $\{E, F, G\}$ is a *nogood*, it cannot be used, and in fact node *N* exists in the network without any justification. It must therefore be marked *OUT*. As a result, the node *R* too has no valid justification, and must also be marked *OUT*.

The algorithm for computing the labels works in an incremental fashion as new nodes are given to the *ATMS* along with justifications. When a new inference is made, the new node is added with an empty label, and a two stage procedure begins.

1. The label of the new node is computed from the labels of its antecedents.
2. Any new labels are propagated across the network.

We illustrate the two stages with an example. Let us say that first the following two nodes are added to the ATMS depicted in Figure 17.21.

$\langle T, \{\{C\}, \{E, F\}\}, \{\{C\}, \{E, F\}\} \rangle$

$\langle U, \{\{B, C\}, \{B, G\}\}, \{\{B, C\}, \{B, G\}\} \rangle$

At this point, let the rule $(T \wedge U \supset V)$ be fired by the reasoner and the node $\langle V, \{\}, \{\{T, U\}\} \rangle$ be added to the network as shown in Figure 17.22. Note that the justification gets filled in from the antecedents in the rule, while the environment is yet empty.

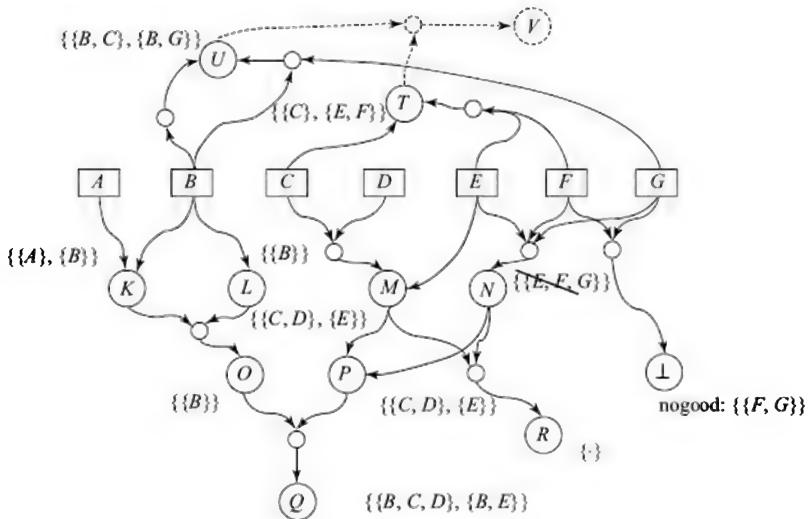


Figure 17.22 After nodes $\langle T, \{\{C\}, \{E, F\}\} \rangle$ and $\langle U, \{\{B, C\}, \{B, G\}\} \rangle$ have been added as shown, node V is just about to be added.

The high level algorithm *LabelsATMS* for determining the labels for the new nodes is given in Figure 17.23. Let there be K antecedents in the justification, and let $\{L_1, \dots, L_K\}$ be the labels of the antecedents, and *NoGs* the set of nogoods stored. The algorithm begins with one label, and incrementally incorporates the labels from the rest of the antecedents. At each step, it replaces each environment by computing unions with each environment of the incoming label. From the resulting set of environments, it discards any that are subsumed by more general ones, and also discards any environments that subsume a nogood.

```

LabelsATMS( $K, \{L_1, \dots, L_K\}, NoGs$ )
1   $L \leftarrow L_1$ 
2  for  $i \leftarrow 2$  to  $K$ 
3       $NewLabel \leftarrow \{ \}$ 
4      for each environment  $E \in L$ 
5          for each environment  $E_{ij} \in L_i$ 
6               $NewLabel \leftarrow NewLabel \cup \{E \cup E_{ij}\}$ 
7       $L \leftarrow NewLabel$ 
8  for each  $E_j \in L$ 
9      for each  $NG \in NoGs$ 
10         if  $NG \subseteq E_j$ 
11             then  $L \leftarrow L \setminus E_j$ 
12  for each  $E_j \in L$ 
13      for each  $E_i \in L$ 
14         if  $E_i \subseteq E_j$ 
15             then  $L \leftarrow L \setminus E_j$ 
16  return  $L$ 

```

Figure 17.23 The procedure *LabelsATMS* accepts a set of K antecedents of a node along with their labels $\{L_1, \dots, L_K\}$, and a set of nogoods $NoGs$, and computes the label L of the consequent node.

In our example, when the node V is added the candidate set of labels generated, after the loop in lines 2 to 7 is executed is, $\{\{B, C\}, \{B, C, G\}, \{B, C, E, F\}, \{B, E, F, G\}\}$. Then in lines 8 to 11, the environment $\{B, E, F, G\}$ is removed because it is a superset of the nogood $\{F, G\}$. After that, the algorithm chooses the minimal environments in lines 12 to 15. In this process, both $\{B, C, G\}$ and $\{B, C, E, F\}$ are discarded since they are supersets of $\{B, C\}$. The algorithm finally returns the label containing one environment $\{B, C\}$. The reader should verify that nodes B and C being *IN* is sufficient for node V being *IN*. Since this is the only environment for V , it is also a necessary condition.

If a label of a node changes then it may have repercussions on the status of other nodes. When any label L for any node N changes, then the ATMS algorithm does the following.

1. If the node N is a contradiction then,
 - (a) Mark all environments in L as nogoods.
 - (b) If any of these environments, or their superset occurs in any node label, then remove it from that label.
2. If N is not a contradiction, then recursively call *LabelsATMS* for all nodes in whose justification the node N occurs as an antecedent.

In practice, using the above algorithm would result in too much redundant work being done. This is because the *ATMS* would receive information, one inference or one justification at a time. It is only the effects of this justification that needs to be propagated to the consequent, instead of re-computing its label again from scratch. The same would apply to propagating changes down the network. Designing the

algorithms for doing only incremental changes is left as an exercise. The interested reader is also referred to (de Kleer, 1986; 1986a; 1986b) and (Forbus and de Kleer, 1992) for a comprehensive study of the design and deployment of variations of truth maintenance algorithms.

17.4.2 The ATMS in GDE and GDE⁺

In the diagnosis problem covered in Section 17.3, the assumptions are the statements about the correct function of components. They refer to statements $\neg Ab(component)$ or $OK(component)$.

As described in the section on Model Based Diagnosis, the rules used by the reasoning engine correspond to statements like,

- Multiplier(M) $\wedge \neg Ab(M) \supset Product(output(M), input_1(M), input_2(M))$
- or $OK(battery) \supset Voltage(T_+) = +$
- or $OK(battery) \supset Voltage(T_-) = -$
- or $(OK(Wire_1) \wedge LWire_1 = T_+ \wedge Voltage(T_+) = +) \supset Voltage(RWire_1) = +$
- or $(OK(bulb) \wedge Voltage(Tbulb) = + \wedge Voltage(Bbulb) = -) \supset On(bulb)$
- or $(OK(bulb) \wedge Voltage(Tbulb) = - \wedge Voltage(Bbulb) = +) \supset On(bulb)$

and so on. The nodes in the *ATMS* constitute of inferred values like $On(bulb_1)$, along with their justifications as well as observed values like $\neg On(bulb_1)$, which are facts or premises. Then rules like,

$$On(bulb_1) \wedge \neg On(bulb_1) \supset \perp$$

lead to contradictions which in turn lead to the identification of nogoods, which are the conflict sets used by the diagnostic engine. Thus, using *the ATMS* makes it possible to identify all conflict sets as more and more observations are made. This is followed by the generation of minimal candidates for diagnosis as described in Section 17.3.3.

The introduction of fault models in Section 17.3.4 brought in relations between the *OK* statements, reproduced again below. Given *F* fault modes for a component *C*,

$$\forall i < F (OK(C) \supset OK_i(C))$$

and

$$OK_1(C) \wedge OK_2(C) \wedge \dots \wedge OK_F(C) \supset OK(C)$$

where $OK(C)$ asserts that component *C* is not broken in any way, while $OK_i(C)$ says that it is not broken in the *i*th fault mode. The universally quantified statement is equivalent to a set of statements of the form

$$OK(C) \supset OK_i(C)$$

This gives us *F+1* implication statements that can form *F+1* justifications in the *ATMS*. However, by themselves, these justifications are circular in nature. One needs to separate the *relation between*

different states of the component—broken in a particular fault mode, or functioning correctly and the *assumptions* about one of these modes. We adopt the convention used in (Struss and Dressler, 1989) that the lowercase letters are used for the assumption about the datum, and the upper case ones for the datum itself. The network of justifications then looks like as depicted in Figure 17.24. Thus for example, the node for $OK_3(C)$ could be labelled *IN*, either because of the assumption $ok_3(C)$ being *IN* or the node $OK(C)$ being *IN*.

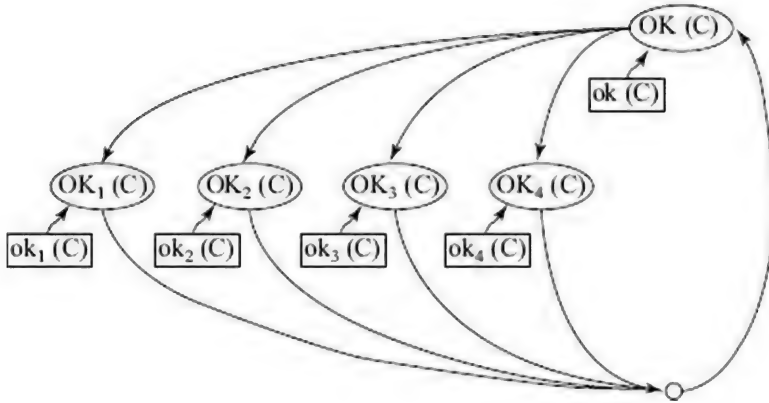


Figure 17.24 Given four fault modes for a component C , the following nodes and justifications are inserted into the ATMS.

A more significant way in which reasoning with fault models is different from working only with models of correctly working components is that one may be required to make an assumption of the form $\neg ok_i(C)$. So far in the diagnosis approach, we have seen the assumptions are only of the form $ok(C)$, and a conflict or a nogood is a collection of such assumptions that cannot be true at together.

Handling negative assumptions requires an extension to the *ATMS* algorithms, as for example proposed in (Dressler, 1988; 1989) and (de Kleer, 1988). The following two rules state that both an assumption and its negation cannot be made together, and also that one of the two has to necessarily be made. Let n_0 be assumption justifying the datum N_0 , and $\neg n_0$ justifying $\neg N_0$. The two rules are,

1. The consistent belief rule. Both a datum and its negation cannot hold together.

$$N_0 \wedge \neg N_0 \supset \perp$$

2. The nogood inference rule. One of n_0 and $\neg n_0$ must hold. If $\{n_1, n_2, \dots, n_k, \neg n_0\}$ is a nogood, then the following justification is added to the *ATMS*.

$$N_1 \wedge N_2 \wedge \dots \wedge N_k \supset N_0$$

Given the above extension to the *ATMS*, one can extend the *GDE* to handle fault models of components. In a system named *GDE+*, Struss and Dressler (1989) show that the knowledge of behaviour of components in faulty modes can be exploited to exonerate certain components. Given a component *C*, if the assumption of all its fault modes is inconsistent with the observed behaviour then that component can be said to be working correctly. Consequently, it cannot be part of a candidate diagnosis, and any candidate that includes *C* can be discarded.

Consider a component *C* whose observed behaviour is inconsistent with all its fault modes. An example of such a component is the bulb *B₃* in Figure 17.16 which is lit, while its only fault mode says that if it is faulty, it cannot be lit. Let us assume that the component *C* has *M* fault modes $\{\neg ok_1(C), \neg ok_2(C), \dots, \neg ok_M(C)\}$. Further, let us assume that it is connected or related to components *A* and *B*, which have *P* and *Q* fault modes, respectively $\{\neg ok_1(A), \neg ok_2(A), \dots, \neg ok_P(A)\}$ and $\{\neg ok_1(B), \neg ok_2(B), \dots, \neg ok_Q(B)\}$. In addition, they all have the *OK* modes *ok*(*A*), *ok*(*B*), and *ok*(*C*).

GDE+ creates the corresponding nodes in the extended *ATMS*. For each fault mode, it adds both the assumptions $\neg ok_i(component)$ and *ok_i(component)* along with the nodes $\neg OK_i(component)$ and *OK_i(component)*. Likewise, for the general descriptions *ok*(*component*) and $\neg ok(component)$.

Let us say that the given set of observations *O* is inconsistent with every faulty mode of the component *C*. This could happen if the fault modes assumptions are inconsistent with all modes, correct and faulty, of the related components, with respect to the observations *O*. Consider a particular fault mode *ok_i(C)*. This means that,

$$\begin{aligned} \forall asmA \in \{ \neg ok_1(A), \neg ok_2(A), \dots, \neg ok_P(A), ok(A) \} \\ \forall asmB \in \{ \neg ok_1(B), \neg ok_2(B), \dots, \neg ok_Q(B), ok(B) \} \\ \{ \neg ok_i(C), asmA, asmB \} \text{ is a nogood.} \end{aligned}$$

That is, the following sets of assumptions are nogoods.

$$\{\neg ok_i(C), \neg ok_1(A), \neg ok_1(B)\}$$

$$\{\neg ok_i(C), \neg ok_1(A), \neg ok_2(B)\}$$

...

$$\{\neg ok_i(C), \neg ok_1(A), \neg ok_Q(B)\}$$

$$\{\neg ok_i(C), \neg ok_1(A), ok(B)\}$$

$$\{\neg ok_i(C), \neg ok_2(A), \neg ok_1(B)\}$$

$$\{\neg ok_i(C), \neg ok_2(A), \neg ok_2(B)\}$$

...

$$\{\neg ok_i(C), \neg ok_2(A), \neg ok_Q(B)\}$$

$$\{\neg ok_i(C), \neg ok_2(A), ok(B)\}$$

...

$$\{\neg ok_i(C), \neg ok_P(A), \neg ok_1(B)\}$$

$$\{\neg ok_i(C), \neg ok_P(A), \neg ok_2(B)\}$$

...

$$\{\neg ok_i(C), \neg ok_P(A), \neg ok_Q(B)\}$$

$$\{\neg ok_i(C), \neg ok_P(A), ok(B)\}$$

$$\{\neg ok_i(C), ok(A), \neg ok_1(B)\}$$

$$\{\neg ok_i(C), ok(A), \neg ok_2(B)\}$$

...

$$\{\neg ok_i(C), ok(A), \neg ok_Q(B)\}$$

$$\{\neg ok_i(C), ok(A), ok(B)\}$$

In particular,

$$\forall j (1 \leq j \leq Q \supset \text{nogood}\{\neg ok_i(C), \text{asm}A, \neg ok_j(B)\})$$

This means that,

$$\forall j (1 \leq j \leq Q \supset (\neg OK_j(C) \wedge asmA \supset OK_j(B)))$$

is added as a justification. And since,

$$OK_1(B) \wedge OK_2(B) \wedge \dots \wedge OK_Q(B) \supset OK(B)$$

it follows that $OK(B)$ must be *IN*. Now since all environments of the form $\{\neg ok_i(C), asmA, ok(B)\}$ are also nogoods, it follows that all environments of the form $\{\neg ok_i(C), asmA\}$ are nogoods. By a similar argument for $1 < j < P$, we can infer justifications of the form $(\neg OK_j(C) \supset OK(A))$, which leads us to the justification,

$$OK_1(A) \wedge OK_2(A) \wedge \dots \wedge OK_P(A) \supset OK(A)$$

If $OK(A)$ is *IN* then the nogood $\{\neg ok_i(C), ok(A)\}$ reduces to $\{\neg ok_i(C)\}$ which results in the conclusion that $OK_i(C)$ must be *IN*. Given the fact that *all* fault modes of component *C* are inconsistent with the observations, we infer a similar statement for all the fault modes. That is,

$$\forall i (1 \leq i \leq M \supset OK_i(C))$$

which in turn leads to the conclusion $OK(C)$. This means that any candidate diagnosis of the form $\{\dots, \neg ok(C), \dots\}$ must be discarded. As shown in the example in Section 17.3.4, this can lead to a considerable amount of pruning in the space of candidate diagnosis.

In that specific example, once we infer that the bulb B_3 is okay, we discard all candidates that include B_3 . Reasoning in a similar fashion with their fault models, one can exonerate the battery S and the wires W_1 to W_6 . This leaves the only minimal candidate which says that the bulbs B_1 and B_2 must be broken.

17.5 Probabilistic Reasoning

So far our view of knowledge and belief has been through a Boolean lens. Propositions are either *true* or *false*. And they take a truth value, due to their relation to other propositions by a process of inference, or because they are assumptions or premises. Our goal has been to arrive at a consistent set of beliefs, in which there are no contradictions.

However, it is not practical to always maintain large amounts of propositional knowledge, and make the relevant logical inferences. For example, one cannot describe the world in terms of the fundamental particles our universe is made of. As discussed earlier (see Section 14.1), one way we get around the problem is to devise abstractions that exist as reified objects that we can reason with in a tractable manner. However, even after doing that, the amount of propositional knowledge remains

unmanageable.

There are other reasons why we cannot make definitive statements about the world, the principal one being that we simply do not have enough information about the world. Consider the statement—"There exists intelligent life elsewhere in our galaxy" or even "There exists life somewhere else out there in the universe" ²⁷. There are *innumerable* questions we are unable to answer exactly (each of them can be converted into a yes/no question or a proposition). How many species exist on our earth? The answers are *estimates* ranging from 2 million to 100 million²⁸. How many green leaves are there on that tree? If white plays perfectly, can it always win in chess? Who leaked the general's letter? It is the monsoon season in Mumbai. Will it rain on July 11? Anrav is playing outside. Has he finished his homework? Anish is looking very happy. Did he win his badminton match? Malala has high fever and shivering. Has she got malaria?

Even though we cannot assign Boolean truth values to such statements, we would often like to make some decisions that depend upon the truth values of these statements, and in the absence of definitive truth values, we have to work with some kinds of estimates.

The way to deal with this is to resort to maintaining *degrees of belief*, represented by numerical values. One way of doing this is by assigning a proposition a *probability* of being *true*. For example, if one hears of a bird named Chirpy then one might say that the probability that Chirpy can fly is (say) 0.8. The actual value for this probability may differ from person to person, based on their individual subjective experiences, and the given context. For example, a person who has never had experiences of being lied to by anyone may assign a high value to the probability of a stranger being honest. Some other may have a conditional belief that the students from a particular school are honest, or that people in a particular locality are prone to tell lies.

Such values are known as *subjective probabilities*. They are also known as *Bayesian* probabilities that encode the degree of belief an agent has in the statement. This is opposed to the *frequentist* view in which the probability values are arrived at by collecting statistical samples.

Consider the task of diagnosis discussed earlier. We had observed in Section 17.3.3 that knowledge of the probabilities of components failing could help decide where to take the next measurement. Let us say we want to know the probability of a particular component *C* failing. Let us also make an assumption that the probabilities will depend on time. Very often, the older a component, the more likely it is to fail. To simplify matters, let us deal with time in intervals and assign a probability value of failure during each of these time intervals. Let us also make the (not so reasonable) assumption that the total life of the component is not more than *N* years, after which we *know* it will fail.

Then we have *N* time intervals $t_1 \dots t_N$ and corresponding to each of these intervals is a numeric value $p_1 \dots p_N$, where p_i denotes the

probability of the component C failing during the i^{th} time interval t_i .

The set of N time intervals define N corresponding propositions. Each proposition F_i asserts that the component will fail in the i^{th} time interval t_i . The set of statements $\{F_1 \dots F_N\}$ is called the *sample space*. A sample space is a set of mutually exclusive (only one can be true) and exhaustive (at least one has to be true) set of statements. Since we do not know which statement is true in fact, the numbers $p_1 \dots p_N$ are used to assign our belief in their chance of being true. That is,

$$P(F_i) = p_i$$

where $P(F_i)$ denotes the probability of proposition F_i being *true*, and is a short form for $P(F_i = \text{true})$. The function P is called the *probability mass function*. For these numbers to be probabilities, they have to satisfy the following properties.

$$0 \leq P(F_i) \leq 1$$

and

$$\sum_{1 \leq i \leq N} P(F_i) = 1$$

The first inequality states that the minimum probability value possible is zero and the maximum possible value is one. The value zero corresponds to total disbelief and the value one corresponds to total belief. Observe that these are measures of belief of the agent and not what is true in the world. In the real world, the component will fail or not fail in a given time interval. If one were to toss a fair coin then one would say that the probability of it falling heads up is 0.5, as is the probability of it falling tails up. But when you do toss a coin, it either falls heads up or tails up, and there is no uncertainty about it any more.

The second property states that the sum of probabilities over the sample space must be one, signifying that the space is exhaustive. This means that all possible events have been covered. In the coin tossing example, the coin will either land heads up or it will land tails up, and one of the two will certainly happen (we ignore the possibility of it standing on its side). These two events form the sample space.

This clearly illustrates that probabilities are measures of belief of agents for facts about which there is some uncertainty. When this uncertainty is removed, the true facts emerge and probability has no more role to play. Given that we are told Chirpy is a bird, we may not know with certainty whether it can fly or not, but we bring our background knowledge to fore and assign a probability to the proposition being true. Later, more information may reveal the actual truth value of the statement. Agents act in the real world, based on their beliefs. Consider the stark example of day trading in the stock markets. One agent sells a stock and the other agent buys it. The seller has a belief that the stock price will not change or will go down. The buyer, on the other hand,

believes that the price will go up. At the end of the day only one of them will be proved correct. A bridge player may try a finesse believing that the probability of success, depending upon which opponent has a given card, is fifty percent. But when the play is made, the layout becomes known.

One can also think of the probabilities as possible values for a *random variable*. In our component failing example, let us say the random variable is called *FailTimeC* and can take values from the different time intervals defined above. Then, the probabilities are described by a set of statements of the form,

$$P(\text{FailTimeC} = t_i) = p_i$$

The ordered set of values $\langle p_1 \dots p_N \rangle$ then defines a *probability distribution* $\mathbb{P}(\text{FailTimeC})$ for the random variable *FailTimeC*. Figure 17.25 depicts a possible probability distribution for *FailTimeC* for a value of $N = 6$. The values are $\langle 0, 0.1, 0.2, 0.4, 0.2, 0.1 \rangle$. The given probability distribution assigns zero probability to the component failing in the first year (time interval), after which, the probability of failing increases for the next two years, peaking at year 4 and then declining.

The figure also shows the cumulative probabilities of the component failing within the longer time interval. The probability that the component will fail in the first K years is given by $\sum_{1 \leq i \leq K} P(\text{FailTimeC} = t_i) = \sum_{1 \leq i \leq K} p_i$. These probabilities are depicted by bars with dashes in Figure 17.25. Observe that the cumulative probability in the last interval is 1.

The above formulation of probabilities of failure of the component is over a discrete space in which the random variable can take values from a discrete space. In practice, many problems involve continuous space. In this example, we can model time as a continuous variable and describe the probability of failure as a function of this real values time. In doing so, the probability distribution is replaced by a

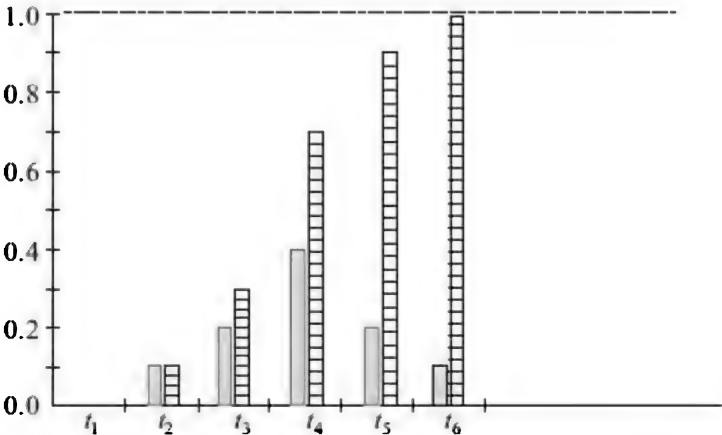


Figure 17.25 A probability distribution over six time intervals. The shaded bars show the probability of failure in each time interval. The bars with dashes depict the cumulative

probability over time.

probability density function. Figure 17.26 depicts a possible density function $P(t)$ for component failure, corresponding to the mass function on Figure 17.25. Observe that we have extended the life of the component indefinitely in the figure.

The probability density function must also satisfy two conditions.

$$P(t) \geq 0$$

and

$$\int_0^{\infty} P(t) dt = 1$$

That is, the area under the curve must be equal to 1.

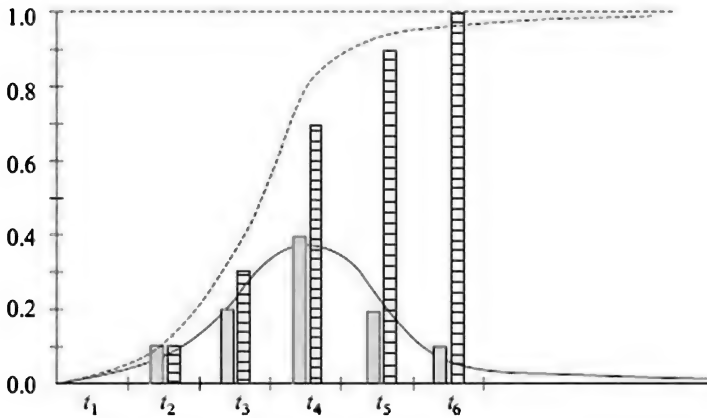


Figure 17.26 With random variables taking continuous values, the probability distribution is replaced by a probability density function. The solid curve above is a plot of this density function, while the dashed curve is the cumulative probability.

The cumulative probability of the component failing within time K is given by the area under curve, $\int_0^K P(t) dt$. Observe that in the figure above the cumulative probability, depicted by the dashed curve, tends to 1, as the time tends to infinity.

In general, the following can be said about probability density functions. They must be positive valued functions over the entire sample space and the integral (or area) of the curve must be 1. To determine the probability that the event will occur, during an interval $<m, n>$ is obtained by computing the area under the curve over the interval $\int_m^n P(t) dt$.

One can meaningfully talk of computing the probability over an interval, however small.

The curves drawn in the Figure 17.26 are hand drawn for illustration. In practice, one would like to define the curve using a set of parameters

and a function definition. How does one arrive at these probability density functions or probability distributions? There are basically two approaches. The frequentist approach is to carry out some kind of experiments, generate a set of data points, and try and estimate the function by a process akin to curve fitting. The subjective approach would be to bring to fore one's knowledge and hypothesize a function, and perhaps learn the parameter values by matching with observed data.

Readers familiar with probability and statistics, would notice the similarity of the curve in Figure 17.26 with the well known Gaussian or normal distribution, characterized by two parameters—the mean value and the standard distribution. There are other distributions as well that have been studied by people working in pattern recognition, machine learning, and many other applications of probability and statistics. We shall not pursue the study of probability distributions, but instead turn our attention to how different random variables influence the probabilities of each other, and how one can make some sort of probabilistic influences.

17.5.1 Conditional Probabilities and Bayesian Reasoning

When we say that the probability of a proposition being true is a certain value, it is in fact backed up by all the world knowledge that the agent has brought to fore. We think of $P(a)$ as the probability of “a” being true. This statement really sums up all our knowledge about the world in a single number. In that sense, it is in fact a short form for $P(a \mid k)$ —read as the probability of “a” being true given “k”—where “k” stands for everything that we know about the world but are not in a position to articulate (Pearl, 1988). In that sense, probabilities are always conditional on what we know already.

In practice, we would like to exploit any information that we receive to revise our belief (probability) estimates. This is akin to making inferences in logic. For example, seeing a big crowd already waiting, one may revise one's chances of quickly finding a seat in a favourite restaurant. Reading a review of a new movie may change our belief of how enjoyable the movie would be. Watching the audience come out of the previous show may further influence our belief. A medical doctor may give greater credence to the belief that a patient is afflicted by a particular disease, if an epidemic has started. As more and more cards are played, a bridge player forms stronger beliefs of how the rest of the cards are distributed. There are many cues that we take to influence our beliefs and decisions.

The first thing one has to do is to decide upon the set of random variables for modelling a given situation. Let $X = \langle X_1, X_2, \dots, X_n \rangle$ be a vector on N random variables. Let each variable X_i have a sample space $D_i = \langle V_{i1}, V_{i2}, \dots, V_{ik} \rangle$ which is a vector of values that the variable X_i can take. Observe that each variable may have a domain of independent size²⁹.

Each such variable may have a probability distribution associated with it, known as the *a priori* probability. Let $\mathbb{P}_i = \langle p_{i1}, p_{i2}, \dots, p_{ik} \rangle$ be the *a priori* probability distribution for the i^{th} random variable.

The task at hand is that given the above *a priori* probabilities, given that some random variables have taken certain values, and some additional knowledge, to determine the *a posteriori* probabilities of other variables of interest.

The additional knowledge alluded to here is knowledge of conditional probabilities. That is knowledge of how variables taking some values affect the probabilities of other variables. The conditional probability of variable X_i being influenced by variables X_m and X_n is written as $\mathbb{P}(X_i | X_m, X_n)$.

It is not necessary that two random variables are conditionally dependent. If probabilities do not depend upon each other, we say that the variables are independent. The simplest example of independent events is coin tosses, or tossing say two coins simultaneously. Let X and Y be the random variables for the two coins, which can take values from the set {heads, tails}. Then we know that,

$$\mathbb{P}(X, Y) = \mathbb{P}(X) * \mathbb{P}(Y)$$

But we also know that by the product rule,

$$\mathbb{P}(X, Y) = \mathbb{P}(X|Y) * \mathbb{P}(Y) = \mathbb{P}(Y|X) * \mathbb{P}(X)$$

From the above two equations, we can derive,

$$\mathbb{P}(X|Y) = \mathbb{P}(X)$$

which can be seen as an equation characterizing the independence of X from Y . The value that Y takes, if tossed earlier, does not influence the probability distribution of X . In a similar manner, we can observe that Y is independent of X .

Given N variables, one can talk of a *joint probability distribution* that assigns a probability to each combination of values the N variables can take. We denote this joint probability distribution by $\mathbb{P}(X_1, X_2, \dots, X_n)$. An element of this probability distribution would be of the form $P(X_1 = V_{13}, X_2 = V_{27}, \dots, X_n = V_{n4}) = 0.06$, which says that the probability that X_1 takes the third value V_{13} from its domain, and X_2 takes its seventh value V_{27} , and so on till X_n is 0.06. Wherever there is no ambiguity, we may write this as $P(V_{13}, V_{27}, \dots, V_{n4}) = 0.06$.

Observe that the joint probability distribution in the general case could be very large. If each of the N random variables can take K values, then there are K^N different elements in the joint probability distribution. Obviously, filling up these values would be a huge task. In the special

case when each random variable corresponds to a sentence in propositional logic, taking the value *true* or *false*, the joint probability distribution will have 2^N entries. Each of these 2^N entries will correspond to an interpretation I of the set of sentences (see Chapter 12). Except that the interpretation assigns *true* or *false* values to each proposition, and the joint probability distribution assigns a measure of belief to each combination of such values. We could think of the probability of each interpretation I being true as $P(I)$, which is the measure of belief of that combination of truth values holding. Then given an arbitrary sentence a in the logic, we could compute the probability of the sentence being true as the sum of the probabilities of all those interpretations, in which a is necessarily *true* (Brachman and Levesque, 2004).

$$P(\alpha) = \sum_{I \models \alpha} P(I)$$

However, note that this approach is likely to be computationally very expensive.

Consider a domain in which there are two random variables X and Y with domains $D_X = \langle x_1, \dots, x_k \rangle$ and $D_Y = \langle y_1, \dots, y_p \rangle$. The two fundamental rules of probability theory are the *sum rule* and the *product rule*. The sum rule is,

$$P(x_k) = \sum_{1 \leq i \leq p} P(x_k y_i)$$

which we can also write as,

$$\mathbb{P}(X) = \sum_Y \mathbb{P}(X, Y)$$

This formula allows us to compute the probability $P(X=x_k)$ from the joint probability distribution. The rule says that, given the joint probability distribution for X and Y , the probability distribution for X , also called the marginal probability, can be computed by adding up the corresponding values for all values of Y .

The product rule allows us to compute the joint probability, given the conditional properties and the marginal properties.

$$P(x_k y_i) = P(y_i | x_k) * P(x_k) = P(x_k | y_i) * P(y_i),$$

or

$$\begin{aligned} \mathbb{P}(X, Y) &= \mathbb{P}(Y | X) * \mathbb{P}(X) \\ &= \mathbb{P}(X | Y) * \mathbb{P}(Y) \end{aligned}$$

The product rule leads us to the well known *Bayes' theorem*.

$$P(y_i | x_k) = P(x_k | y_i) * P(y_i) / P(x_k)$$

or

$$\mathbb{P}(Y | X) = \mathbb{P}(X | Y) * \mathbb{P}(Y) / \mathbb{P}(X)$$

Given a set of random variables, what do the conditional probabilities represent? Given the variables X and Y , the expression $\mathbb{P}(Y | X)$ defines the probability distribution of X , given values of the variables of Y . For example, we could say that $P(\text{Fruit} = \text{Litchi} | \text{Season} = \text{Summer}) = 0.7$ to assert that if the season is summer, the probabilities of one finding litchis in the market is 0.7. One could also say that $P(\text{Fruit} = \text{Litchi} | \text{Season} = \text{Summer}, \text{City} = \text{Dehradun}) = 0.9$, if one is in Dehradun in the summer then the probability is even higher. If one had conditional probabilities for other cities and other seasons, then one could compute $P(\text{Fruit} = \text{Litchi})$ by using the sum rule and marginalising the other variables.

Let us look at a small contrived example. In the following example, the probabilities are all with respect to your friend Sumedha being in one of the cities or eating the fruit. Let us assume that Sumedha can be found in one of three cities, Bengaluru with probability 0.6, Dehradun with probability 0.2 and Madurai with probability 0.2. Also let us say that the conditional probabilities for finding (and eating) litchis in the cities are as follows,

$$P(\text{Litchi} | \text{Bengaluru}) = 0.4$$

$$P(\text{Litchi} | \text{Madurai}) = 0.4$$

$$P(\text{Litchi} | \text{Dehradun}) = 0.9$$

You see Sumedha's status message and find that it says "Enjoying litchis". Where do you think that she is likely to be?

Let us first apply the sum rule to compute $P(\text{Litchi})$, which stands for the probability that Sumedha is eating litchis.

$$\begin{aligned} P(\text{Litchi}) &= P(\text{Litchi} | \text{Bengaluru}) * P(\text{Bengaluru}) \\ &\quad + P(\text{Litchi} | \text{Madurai}) * P(\text{Madurai}) \\ &\quad + P(\text{Litchi} | \text{Dehradun}) * P(\text{Dehradun}) \\ &= 0.4 * 0.6 + 0.4 * 0.2 + 0.9 * 0.2 = 0.5 \end{aligned}$$

The overall probability of her eating litchis is 0.5. Now knowing that Sumedha is in fact eating litchis, we can apply the Bayes' rule to compute the following conditional probabilities.

$$\begin{aligned}
 P(\text{Bengaluru} \mid \text{Litchi}) &= P(\text{Litchi} \mid \text{Bengaluru}) * P(\text{Bengaluru}) / P(\text{Litchi}) \\
 &= 0.4 * 0.6 / P(\text{Litchi}) \\
 &= 0.24 / P(\text{Litchi}) \\
 &= 0.24 / 0.5 \\
 &= 0.48
 \end{aligned}$$

$$\begin{aligned}
 P(\text{Madurai} \mid \text{Litchi}) &= P(\text{Litchi} \mid \text{Madurai}) * P(\text{Madurai}) / P(\text{Litchi}) \\
 &= 0.4 * 0.2 / P(\text{Litchi}) \\
 &= 0.08 / P(\text{Litchi}) \\
 &= 0.08 / 0.5 \\
 &= 0.16
 \end{aligned}$$

$$\begin{aligned}
 P(\text{Dehradun} \mid \text{Litchi}) &= P(\text{Litchi} \mid \text{Dehradun}) * P(\text{Dehradun}) / P(\text{Litchi}) \\
 &= 0.9 * 0.2 / P(\text{Litchi}) \\
 &= 0.18 / P(\text{Litchi}) \\
 &= 0.18 / 0.5 \\
 &= 0.36
 \end{aligned}$$

The three conditional probabilities $P(\text{Litchi} \mid \text{Bengaluru})$, $P(\text{Litchi} \mid \text{Madurai})$, $P(\text{Litchi} \mid \text{Dehradun})$ are the *likelihoods* of the variable $\text{Fruit}=\text{Litchi}$, given the three values of random variable City . The expressions on left hand side of the *posterior* probabilities of each city, given the evidence that litchis are available. They are 0.48, 0.16 and 0.36 for Bengaluru, Madurai and Dehradun respectively. These are also called the *a posteriori* probabilities, as compared the *a priori* probabilities we started with, 0.6, 0.2 and 0.2 respectively. The reader should verify that the three probabilities add up to 1 in both cases, signifying that the Sumedha *has* to be in one of the three cities. The *prior* probabilities were revised to the *posterior* probabilities after getting the *evidence* (of eating litchis). In machine learning literature, the City variable would define the *hypothesis* space and this approach computes the *maximum a posteriori* (MAP) hypothesis (Mitchell, 1997). Observe that we do not really need to compute $P(\text{Litchi})$, which is a common factor in all the likelihood estimates, and we could have chosen the maximum of $P(\text{Litchi} \mid X) * P(X)$, with X taking the values of the three cities.

If we did not have *a priori* information on where Sumedha is likely to be, we could have assumed that the probabilities are equal, and discarded them from our computation. Then we would have simply selected the hypothesis (city) based on where litchis are most likely to be available. This is known as the *maximum likelihood hypothesis*.

The reader would have noticed that despite the fact that litchis are more likely to be available in Dehradun (0.9), one still concluded that Sumedha is likely to be in Bengaluru (even) after hearing that she ate litchis. This was influenced by the predominantly high *a priori* probability of her being in Bengaluru.

Probability calculations can often lead to conclusions that are often counter intuitive. The author has had many an interesting time discussing problems like the Monty Hall problem, the three prisoners problem and the birthday clash problem (see Exercises 18, 19 and 20).

17.5.2 Diagnosis vs. Prediction

We have seen above that the Bayes' rule allows us to express conditional probabilities in a symmetric manner. That is,

$$\mathbb{P}(Y | X) * \mathbb{P}(X) = \mathbb{P}(X | Y) * \mathbb{P}(Y)$$

Given the value of any of the two variables, we can determine the probability of the other one. One of the situations when this is used extensively is in the problem of classification based on evidence, or the problem of diagnosis.

From the perspective of classification, one gets to see some evidence E and one has to pick the most likely hypothesis H . Given the evidence E , the belief accorded to the hypothesis H is given by,

$$\mathbb{P}(H | E) = \mathbb{P}(E | H) * \mathbb{P}(H) / \mathbb{P}(E)$$

The reason why we choose this direction, given the symmetric nature of the product rule, is that this captures the direction of causal behaviour. In the real world, the cause (that is hypothesized) results in behaviour (symptoms) and not the other way around. For example, getting malaria results in high fever and shivering. The disease causes the symptoms and this relation can be captured with greater fidelity. Observe that $P(\text{HighFever} | \text{Malaria}) = 0.97$ is equivalent to say that (Malaria ... HighFever) in the logical framework but with a degree of belief less than 1.

Thus, if the hypothesis space has (say) three possibilities (h_1 , h_2 and h_3) and we have seen some evidence E then we can compute the products as in the previous section and choose the *maximum a posteriori* hypothesis as one with the highest product,

$$h_{\text{MAP}} = \text{argmax}_{h \in H} (P(h|E) * P(E))$$

If the number of variables that constitutes the evidence is many, we could represent them by E_1, E_2, \dots, E_n . Then the maximum likelihood hypothesis would be,

$$h_{\text{MAP}} = \text{argmax}_{h \in H} (P(h|e_1, e_2, \dots, e_n) * P(e_1, e_2, \dots, e_n))$$

which by the product rule would become,

$$h_{\text{MAP}} = \text{argmax}_{h \in H} (P(h|e_1, e_2, \dots, e_n) * P(e_1 | e_2, \dots, e_n) * P(e_2 | e_3, \dots, e_n) * \dots * P(e_n))$$

This leads us towards the Naïve Bayes Classifier that we look again in Chapter 18.

It is claimed that probability theory was devised to address problems in gambling in which players are concerned with odds (Stewart, 2002).

Odds represent the amount the player is willing to wager, based on her belief of the probability of the event being bet on. For example, one would give 1:1 odds that a fair coin will come up heads, and give 1:3 odds that a random card drawn from a standard pack would be a spade. When we say that the odds are $m:n$, this corresponds to the probability $m/(m+n)$ (Jeffrey, 2004). Thus, the odds 1:3 for drawing a spade card correspond to the probability $1/(1+3) = 1/4$, which is the probability of drawing a spade. The odds represent a fair bet, which would even out in the long run.

Judea Pearl (1988) portrays the Bayes' rule using odds as follows.

Let H be the hypothesis we are interested in and let $\neg H$ be the case that H is not true. That is $\neg P(H) = 1 - P(H)$. The prior odds $O(H)$ that H is true are given by,

$$O(H) = P(H) / P(\neg H) = P(H) / (1 - P(H))$$

The posterior odds given the evidence e are defined as $O(H|e)$ where,

$$O(H|e) = P(H|e) / P(\neg H|e)$$

which by applying the Bayes' rule becomes,

$$O(H|e) = \frac{P(e|H) * P(H)}{P(e|\neg H) * P(\neg H)} = \frac{P(e|H) * P(H)}{P(e|\neg H) * (1 - P(H))}$$

If we define the likelihood ratio $L(e | H)$ as,

$$L(e | H) = P(e|H) / P(e|\neg H)$$

then we get the posterior odds,

$$O(H|e) = L(e|H) * O(H)$$

Thus, the *posterior* odds of the hypothesis H being true is the product of the likelihood ratio of the evidence being seen if H were true versus if H were not true, and the *prior* odds of H . The likelihood ratio provides *diagnostic* support given the evidence, and the prior odds are a *predictive* support based on previous knowledge.

Very often when two random variables or propositions are related there exists a *causal* relation between them. For example consider the statement, "A new toy makes a child happy". In a logic framework one may express this as $(\text{NewToy} \supset \text{Happy})$. Then if we add the sentence "NewToy" *Modus Ponens* (see Chapter 12) allows us to add "Happy" as well. If one wants to cater to uncertainty, one might try some variation of default reasoning. In Bayesian reasoning, one expresses this relation as a high conditional probability, for example, $P(\text{Happy} | \text{NewToy}) = 0.8$.

While one can think of the conditional probability as an equivalent rule, one cannot extend the analogy with logic further. This is because logic deals with sets of (true) sentences and the application of a rule like *Modus Ponens* changes the set by adding new sentences to the

knowledge base³⁰. Logic employs an inference engine to add new statements. Given the “fact” that “the child got a new toy”, one can add the sentence “the child is happy.” One cannot do something similar in Bayesian reasoning because probability theory deals with beliefs and not facts. And, in fact, the conditional probability statement already states the conclusion that, given *NewToy*, the probability of *Happy* is 0.8. Consider the following joint probability distribution.

Table 17.12 A small joint probability distribution

	<i>NewToy</i> = <i>yes</i>	<i>NewToy</i> = <i>no</i>
<i>Happy</i> = <i>yes</i>	0.4	0.3
<i>Happy</i> = <i>no</i>	0.1	0.2

The joint probability distribution captures all our beliefs pertaining to the set of random variables. There is nothing new to be discovered. Inspecting the joint probability distribution, one can make the following observations,

$$P(\text{NewToy} = \text{yes})$$

$$P(\text{Happy} = \text{yes})$$

$$P(\text{NewToy} = \text{yes}, \text{Happy} = \text{yes})$$

$$P(\text{Happy} = \text{yes} \mid \text{NewToy} = \text{yes})$$

$$= 0.4 + 0.1 = 0.5$$

$$= 0.4 + 0.3 = 0.7$$

$$= 0.4$$

$$= 0.4 / (0.4 + 0.1) = 0.8$$

As one can see that computing conditional probabilities is done by restricting the counting to that part of the table that corresponds to the given values of the given variables. However, very often we *do not* have the complete joint distribution. We have already observed that this table can be prohibitively large in size. Instead, we have fragments of the table captured in terms of some known *prior* probabilities, and some conditional properties. Computation is needed to determine the *posterior* probabilities of some variables of interest.

Note that the probability of getting a new toy $P(\text{NewToy} = \text{yes})$ is 0.5. Bayesian reasoning does not allow us to change this and assert a “fact” like “*NewToy* = *yes*” and then make an inference about the probability of the child being happy. This would amount to changing our *belief* about the probability of getting a new toy. And these beliefs would have been formed via experience either subjectively, or via conducting a set of experiments from a frequentist perspective. The inference that we can make with a rule like *Modus Ponens* is latently captured in the joint probability statement,

$$P(\text{Happy} = \text{yes} \mid \text{NewToy} = \text{yes}) = 0.8$$

The statement itself captures the essence of reasoning done with Modus Ponens and says the *given* that *NewToy* = *yes*, the probability of *Happy* = *yes* being true is 0.8. The analogy is applicable because $P(\text{Happy} \mid \text{NewToy})$ is an (almost) true statement. Here, we are making a prediction that if a new toy is given to a child, then the child will be happy because new toys make children happy.

On the other hand, one might know that the child is happy, and might be curious about what has made the child happy. This is the problem of diagnosis. Let us say that there are several reasons a child could be happy—it could be an inherently happy nature the child has, or due to the joy of visiting an uncle, or having read a new book, eating an ice cream, etc. (and this would need an appropriately sized, multidimensional joint probability table). Each such case becomes a hypothesis for our investigating agent, and the application of the Bayes' rule as described earlier comes to the rescue. The interested reader should compare the application of the Bayes's rule for determining the MAP hypothesis, with backward chaining in logic described in Chapter 12. In that sense, Bayesian reasoning gives us a sound form of diagnosis.

Diagnosis and prediction address different kinds of uncertainty. Prediction is concerned with events that are to happen. Then a probabilistic statement talks about how likely it is for the event to happen. The inference is aligned with the direction of causation. Diagnosis on the other hand, is the process of unravelling the hidden but existing state of the world, which may be shrouded in uncertainty. A medical doctor may begin with several hypotheses when a patient first consults her. But as she gets more evidence, by means of tests or queries, she homes in on the actual disease or affliction the patient is suffering from.

In the world, the effect follows the cause. Diagnosis involves determining the cause knowing the effect. Sometimes, we form associations between the effect and a cause as a rule or conditional probability. For example, we might say $P(\text{Malaria} \mid \text{HighFever}) = 0.6$, saying that whenever we see high fever we can conclude with probability 0.6 that the patient has malaria. On the surface this looks again analogous to the application of *Modus Ponens* which is a sound rule of inference,

$$\begin{array}{l} \text{HighFever} \\ \text{HighFever} \supset \text{Malaria} \\ \therefore \text{Malaria} \end{array}$$

But in fact it is not the case, because the causal relation is between Malaria and HighFever and not the other way round. If one were to convert this to a logical relation, one might say that (Malaria ... HighFever) is *true* but (HighFever \supset Malaria) is *false*. In that sense, inferring the disease from symptoms is in fact doing the following inference.

HighFever

Malaria \supset HighFever

\therefore Malaria

This inference is called *abduction* and is not a sound rule of inference. This means that if a patient has high fever, one cannot conclusively say that she has malaria. It could be something else too. Nevertheless, this form of reasoning is prevalent amongst human users. Doctors routinely look at symptoms and diagnose the cause. In doing so, they no doubt bring to fore their vast experience³¹, and it is not surprising that patients flock to “reputable” doctors. But the inexperienced ones can go wrong.

Reasoning systems that employ probabilistic knowledge have sometimes tended to mimic the operation of a rule based system, adding new sentences and assigning degrees of belief to them. This becomes an attractive option, given the paucity of probability data. One of the first expert systems developed—MYCIN (Buchanan and Shortliffe, 1984)—in fact expressed knowledge in (IF Symptoms THEN Disease) form, as illustrated below.³²

RULE 156

IF: 1. The site of the culture blood, and

2. The gram stain of the organism is gramneg, and

3. The morphology of the organism is rod, and

4. The portal of entry of the organism is urine, and

5. The patient has not had a genito-urinary manipulative procedure, and

6. Cystitis is not a problem for which the patient has been treated.

THEN: There is suggestive evidence (.6) that the identity of the organism is e.coli

This is possible because we can make a probabilistic statement that “if you see the symptoms then you can say what the disease is”. MYCIN uses certainty factors to express how strong the association is. The certainty factor of a hypothesis H given an evidence E is made up of two values—the measure of belief in H given E written as $MB[H, E]$ and a measure of disbelief in H given E written as $MD[H, E]$. The certainty factor $CF[H, E]$ is defined as,

$$CF[H, E] = MB[H, E] - MD[H, E]$$

The interval $\langle MD, MB \rangle$ for a hypothesis H , given the evidence E , is similar to the confidence interval used in Dempster Shafer theory discussed in Section 17.6.1, except that in MYCIN’s case, these numbers

are provided by the human experts assigning strength to the associations.

One important reason why using *maximum a posteriori* technique is preferable to direct abductive reasoning is that the conditional probability $P(\text{symptom} \mid \text{disease})$ is local in nature, in the sense that given that the patient is known to be afflicted by a particular disease (like malaria), the symptoms (like high fever) do not depend upon other random variables that might be in the model. On the other hand, even if we have the probabilities for $P(\text{disease} \mid \text{symptom})$, the values are likely to depend upon other factors for example whether the patient has had the disease earlier or not (like for chicken pox) or whether there is an epidemic of the disease or not.

Even in everyday life, we use abduction quite effectively. You look at the colour of the *dosa* cooking on the *tava* to decide whether it is done. One sees a person staggering and concludes that he is drunk. But he could be hurt too. You see smoke coming out of a building and infer that there is a fire. One looks at wet grass and concludes that it rained last night.

17.5.3 Propagating Probabilistic Inferences

One can have a knowledge base of conditional probabilities capturing associations between various propositions. One can even think of chaining together such inferences. For example, knowing that her dad's favourite team has won the game, a child might figure that her dad is *likely* to be in a good mood, and *therefore* might decide that that would be a good time when he is *likely* to order her favourite *paneer butter masala* from the neighbouring restaurant.

However, when predictive and abductive rules are mixed up, there could be problems in *chaining* rules to make sequences of inferences. Let us look at the example used by Pearl (1988). Consider the following two rules, with some measure of probability assigned to each inference.

If (the sprinkler was on last night) then (the grass is wet).

If (the grass is wet) then (it rained last night).

The two rules by themselves are fine, and may be used with some benefit. The first one is predictive and the second abductive. But when we try to chain them together, we end up with a rule that says that,

If (the sprinkler was on last night) then (it rained last night).

Clearly there is a problem lurking here. Since one can have both abductive and predictive relations thrown together in a network (see the following section), there could be a danger if two nodes reinforcing themselves in a cyclic fashion (Lowrance, 1982). A hypothesis would

make the evidence appear more likely and vice versa and a belief propagation system could run into trouble. The key to addressing this problem is to keep track of the source of propagation. Every node maintains a *support list* of other nodes in addition to a belief measure. This is reminiscent of storing justifications in the ATMS (see Section 17.4). An alternative approach called *conditioning*, relies on modifying the underlying connectivity of the network to eliminate loops (Pearl, 1985).

If one implements a rule like system in which there is a predictive (malaria implies high fever) abductive (high fever suggests malaria) cycle, it could go into loops amplifying a small bit of evidence. Systems like MYCIN that operate in a modular rule baselike fashion, restrict themselves to abductive reasoning only. In doing so, however, one has to forgo the benefits of prediction. One such benefit is the *explaining away* that can be done with predictive reasoning. For example, if malaria implies fever and if a viral infection can lead to fever too, then finding that the patient has a viral fever explains the fever, and makes belief that the patient has malaria less credible.

We look at an interesting example given by Judea Pearl (1988) that highlights this kind of interaction between inferences. Suppose you have an alarm system installed in your home, and you get a call from your neighbour that the alarm is ringing. You are about to conclude that your house has been burgled and rush home when you happen to glance at your monitor³³ which is showing news that there has been an earthquake in the region. You remember that the last time there was an earthquake too, the alarm had gone off, and this decreases your belief in the possibility of a burglary having happened. Now consider what has happened. If you had a rule ($\text{alarm} \supset \text{burglary}$), then if your belief in alarm goes up, then “normally” your belief in burglary should go up too. But the opposite is happening here. The news of the earthquake has confirmed the fact that the alarm has sounded, but at the same time alleviated your fears that a burglary might have happened.

A variation is that the neighbour may be unreliable, having a tendency for tasteless pranks. You would then have to gather more evidence that the alarm actually sounded. You could try another neighbour but she might be having loud music playing in her house and not quite sure. If you did get some response from her you would still have to worry about combining the evidences together, in a scenario where there may not be too much data on the conditional probabilities needed.

This pull between modularity and reasoning under uncertainty is reflected in other places as well. Recall (Section 17.1) that while doing default reasoning, one has to minimize certain sets in the interpretation, and remove those formulas that that were not entailed by the *KB*. That could require one to look at the complete knowledge base as well. This is closer to the probability approach in which one has to *sum up* the entire knowledge (base). Logical reasoning by itself is modular. The antecedents and the rule imply the consequents.

One difference between logical reasoning and probabilistic

conditioning is that in logic there is an element of locality. For example, if one has the set of sentences,

$$S$$
$$P \supset Q$$
$$Q \supset R$$

there is no difficulty in chaining the inferences because each rule has a local scope. The only thing one needs to infer Q is that S must be true, and in turn when Q is true, it is enough to conclude that R is true. Using probabilities on the other hand when we say that,

$$P(Q|S) = 0.8$$

$$P(R|Q) = 0.7$$

we are really saying that,

$$P(Q|S, K) = 0.8$$

$$P(R|Q, K) = 0.7$$

where K captures all the unarticulated knowledge that contributed to assigning the values. And given S , it is not entirely clear how the probability of R being true would depend upon S being given, even though there appears to be a chain of reasoning. What we really need is the value for the conditional probability $P(R|Q, S)$. Given that we do not have that, and do not know with certainty that $Q=\text{true}$, one approach would be to compute $P(R)$ as a weighted average as follows,

$$P(R) = P(R|Q) * P(Q) + P(R|\neg Q) * P(\neg Q)$$

but this would need more data from the joint probability distribution. If we now take some new evidence E into account, then we would need to compute the posterior probabilities,

$$P(R|E) = P(R|Q, E) * P(Q|E) + P(R|\neg Q, E) * P(\neg Q|E)$$

but this changes the problem to a completely new one.

17.5.4 Belief Networks

Given that the probability $P(a)$ assigns a numerical value to the belief in the proposition " a " *given everything the agent knows about the world*, if one wants to relate the truth value of a statement to the truth values of other statements, one must take into account all other statements that could possibly influence the given statement. Thus, one has to consider the joint probability of all such variables. Given the joint probability distribution, one can compute the marginal and conditional properties of

different variables. However, constructing the joint probability distribution could be a humongous task, and the computations to be done to derive specific probabilities proportionally expensive.

Human beings, on the other hand, make probabilistic observations very quickly, especially on the relatedness of different variables. For example, most people would agree that the appearance of dark clouds *bodes* a (welcome) rain shower. Similarly, they would agree that the fall of a coconut from a tree in Kerala has *no bearing* on the possibility of snow in Manali³⁴. And we do this without recourse to storing huge joint probability tables and enormous amounts of computation.

Belief networks are an attempt to marry probability theory with networks used in various forms for knowledge representation, and the application of propagation techniques over networks for making inferences. There are two types of probabilistic network based representations that are popular. Markov networks are undirected graphs in which an edge between two nodes represents conditional dependence between the two nodes, which stand for random variables. Bayesian belief networks are directed acyclic graphs in which a directed edge captures a *causal relation* between two random variables. Observe that this would still allow for abductive inferences.

Given a set of random variables, there could be varying degrees of conditional dependence between them. If one can identify the dependence relations then one could take recourse to probabilistic graphical models. An example of such models is Bayesian networks, also known as Bayesian Belief Networks (BBNs). Bayesian networks are directed graphs in which edges capture the causal relations between the nodes (random variables).

Importantly, Bayesian networks allow us to get an insight into the *conditional dependencies* between random variables. Consider the randomly generated graph of seven variables $X_1 \dots X_7$ shown in Figure 17.27.

The conditional dependencies between the variables can be determined by looking at the graph. Each variable is causally influenced by its parent. Consequently, the joint distribution of the seven variables can be written as,

$$\mathbb{P}(X_1, \dots, X_7) = \mathbb{P}(X_1) * \mathbb{P}(X_2) * \mathbb{P}(X_3 | X_1, X_2) * \mathbb{P}(X_4 | X_2) * \mathbb{P}(X_5 | X_3) * \mathbb{P}(X_6 | X_3, X_4) * \mathbb{P}(X_7 | X_6)$$

Observe that this equation says that variables X_1 and X_2 are independent of other variables, X_3 is conditionally dependent on X_1 and X_2 only, and so on. This is in lieu of the general expansion for conditional probabilities,

$$\mathbb{P}(X_1, \dots, X_7) = \mathbb{P}(X_1 | X_2 \dots X_7) * \mathbb{P}(X_2 | X_3 \dots X_7) * \dots * \mathbb{P}(X_6 | X_7) * \mathbb{P}(X_7)$$

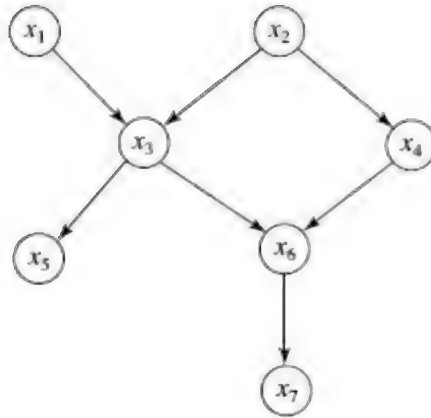


Figure 17.27 A Bayesian network of seven variables.

The latter is simply derived from the product rule *blindly*³⁵ without paying any attention to which variables are causally influencing which other variables. The Bayesian network, on the other hand, clearly marks specific direct influences and is presumably constructed by exploiting domain knowledge.

In general, a Bayesian belief network over N variables is characterized by its joint distribution, defined as the product of the probabilities of all its variables conditioned on their parents,

$$P(X_1, \dots, X_N) = \prod_{k=1}^N P(X_k | \text{Parents}_k)$$

Given the Bayesian network and the conditional probabilities between the linked variables (in both directions), one could estimate the probabilities of some unknown variables given some known variables. In this context, the notion of conditional independence is very valuable, and fortunately can be determined simply by inspecting the state of the network and its observed variables. The statement of conditional independence is as follows. Let X , Y and Z be three random variables and let x , y and z represent the three values that the variable can take. Then if the conditional distribution of X given the value of Z does not depend upon Y , depicted by,

$$P(x | y, z) = P(x | z)$$

we say that the X is conditionally independent of Y given Z . In other words, if one knows the value of the variable Z , then the probability of any value of the variable X does not depend upon the probability of any value of Y . We write this property as $I(X, Z, Y)$ as in (Pearl, 1988). The relation is symmetric. That is, $I(X, Z, Y) \equiv I(Y, Z, X)$. The notation $X, Y \perp\!\!\!\perp Z$

due to Dawid (1979) is also used. In the special case when the two variables are unconditionally independent, like two coin tosses, then we write $I(X, \Phi, Y)$. The relation can be extended to sets of variable $\{X\}$, $\{Y\}$ and $\{Z\}$.

Another way of looking at conditional independence is as follows. Consider the joint probability of X and Y , given the value of variable X . We then have,

$$\begin{aligned} P(x, y | z) &= P(x | y, z) * P(y | z) \quad \text{by the product rule} \\ &= P(x | z) P(y | z) \quad \text{by conditional independence.} \end{aligned}$$

This says that if we know the value of Z then the joint probability of X and Y is simply the product of their marginal properties.

An interesting feature of BBNs is that many such conditional relations can be seen from the graphical point of view. This is based on the notion of separating the two (sets of) nodes by the nodes on which they are conditioned. This property is known as d-separation, where the “d” stands for “directed” (Pearl, 1988). In general, one is interested in knowing whether a node X is influenced by node Y , when the path between the two nodes passed through the node Z . We consider three cases corresponding to the three possible configurations of the arrow directions between the three nodes (see also (Bishop, 2006)).

Case 1

When both X and Y are children of Z in the graph depicting the Bayesian belief network. This implies that both X and Y are causally influenced by Z . An example of such a relation is where Z is the age of a child, X is her height and Y is her reading ability (from (Pearl, 1988)). In the graph, this will be depicted as shown in Figure 17.28a. If we do not know the age of the child then we can relate her height to her reading ability. This is indicated in the graph from the fact that there is a path from the node *height* (X) to the node *reading ability* (Y). But once we know the value of Z , that is the *age* of the child, the other two variables *height* and *reading ability* become independent of each other.

We say that once the node Z is known, it blocks the path from X to Y . In general, if there had been more paths between X and Y then all such paths would have to be blocked to achieve conditional independence between X and Y . Let us say that the set of nodes $\{Z_1, \dots, Z_g\}$ in the graph blocked all paths between X and Y , then we would say, $I(X, \{Z_1, \dots, Z_g\}, Y)$.

Case 2

When Z is a child of X in the graph and Y is a child of Z . Here X could be the result of a match which causally influences the mood of the father which in turn decides whether he would agree to order his daughter's

favourite food. This is depicted in Figure 17.28b with the three variables *match result*, *mood* and *agree to order food*. Again, we can see that once we know that *mood* = *good*, the probability of ordering the food becomes independent of the match result. Another example would be if X = *company performance*, Z = *demand for shares* and Y = *stock price*. Or X = *monsoon status*, Z = *wheat crop production* and Y = *price of wheat*.

In this case too, knowing the value of variable Z blocks the path between X and Y .

Case 3

When both X and Y causally influence node Z . Consider a game in which a player throws two dice and a bell rings whenever some desired result is achieved (for example, the two numbers that show up are the same)³⁶. Now in general, the two dice may show up with numbers independently. Consequently, in general, we should consider the path between them via *bell* as blocked. However, if we knew the value of the variable *bell* then the two throws are not independent anymore. If *bell*=*ringing* then we know the two numbers must be the same. On the other hand, if *bell*=*silent* then we know that they cannot be equal (and therefore are not independent).

In case 3 in Figure 17.28c, when both the arrows are directed toward node Z , the situation is reversed. In the general nodes, X and Y are independent. But if we know the value of Z , they become dependent. Thus, knowing the value of Z *unblocks an initially blocked path*.

Not only the value of Z , but that of its causal descendants had the same effect. Assume for example in our dice game if the bell rings then a waiter appears with a cake as a prize. Knowing that the waiter has appeared with the cake has the same effect of unblocking the path between X and Y . In the network in Figure 17.27, this would mean that knowing the value of X_7 renders X_3 and X_4 dependent.

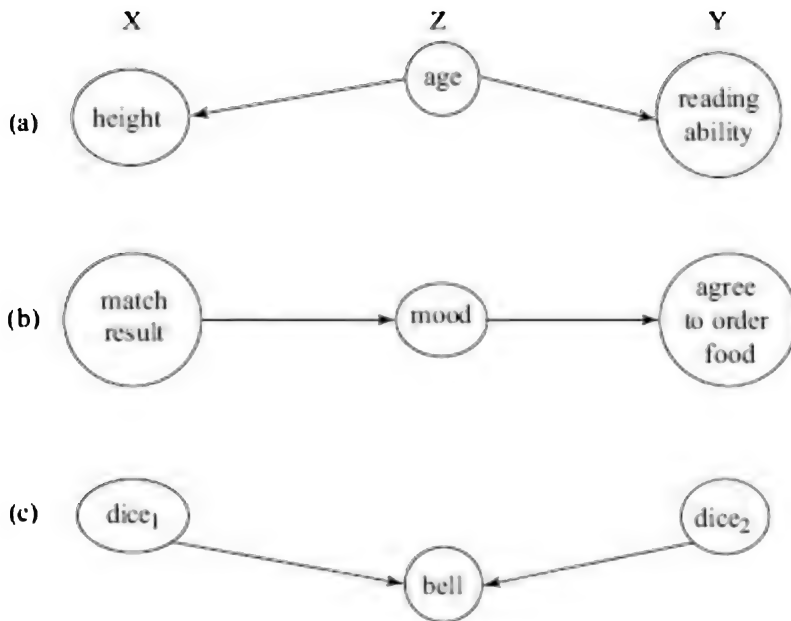


Figure 17.28 The three possible causal relations.

Bayesian networks have been very popular in the area of pattern recognition and machine learning. The idea is similar to the kind of inferences in other network models. One freezes the values of some nodes in the network (the observations) and the task is to determine the posterior probabilities of other nodes in the network, via propagation through the edges.

We shall look at some of the inferences that one can do with Bayesian networks in Chapter 18.

17.6 Stochastic Actions

Another form of uncertainty is when actions are not deterministic. The planning algorithms we studied in Chapters 7 and 10 worked with planning operators which have well defined, deterministic effects. That implies that if the agent plans for an action “a” to be applied in a state S , then the agent is sure that the resulting state S' is defined by (see Section 7.2),

$$S' \leftarrow \{S - effects^-(a)\} \cup effects^+(a)$$

The agent can then assume that it is in state S' and continue planning from there.

However, there are many domains in which the actions are not deterministic, which implies that the agent cannot project the actions into

the future to discover the action sequences that work. The simplest examples of such actions are tossing a coin or throwing a dice. These are, in fact, random number generators, and one cannot predict the outcome deterministically. Other actions that have such a stochastic nature are throwing a basket ball towards the hoop; drawing a card in a game of blackjack; taking a step forward on slippery ice on a glacier; dialling a phone number (could result in a ring or an engaged tone); buying a lottery ticket; and making a “final bid” while haggling over the price of a T-shirt; and so on.

17.6.1 Markov Decision Processes

How does an agent plan in such situations? In classical forward state space planning, one chooses an action and assumes that one goes to a new state from where one can plan again. When the actions are stochastic then one cannot assume that one will go to a particular state. Instead, one may have to consider a *set* of states in any one of which one can end up in with a certain nonzero probability. Would one need to choose actions from each of these states? That is what in fact an approach to planning with Markov Decision Processes (MDPs) does. Only one does not call the output of such reasoning a *plan* anymore, but a *policy*. In this section, we take a brief look at MDPs. Readers interested in greater detail, finer nuances, and theoretical analysis are referred to (Mausam and Kolobov, 2012).

MDPs replace plans with policies. A (complete) policy π is a statement of intent for *every* state in the state transition system. If S is the state space on which the agent operates and A is the set of actions available then

$$\pi: S \rightarrow A$$

The policy specifies an action in *each* state. A partial policy may specify actions for only some subset of the states. The process is a *Markov process*³⁷ because the choice of action depends *only* on the state the system is in, and not on the previous history of states the agent was in. The domain of operation is stochastic because each action $a \in A$ is stochastic. That is, the action does not result in transition to a new state deterministically. Instead, the system may move to a new state s' with a certain probability given by $P(s, a, s')$, where $s, s' \in S$. The probability distribution must satisfy the condition for a given state $s \in S$ and an action $a \in A$,

$$\sum_{s' \in S} P(s, a, s') = 1$$

That is, the action *does* result in a state. The agent could have several types of goals that it wants to achieve. These could be the classical goal

of reaching a final goal state and it could have some trajectory preferences of going through or avoiding certain states. However, since the agent cannot deterministically control its actions but works with a policy that is defined over the entire state space, one needs to specify its performance in terms of evaluating the entire policy. This is done by specifying a *reward function* $R: S \rightarrow [-Large, +Large]$ that denotes the reward obtained by being in a given state. Here, *Large* is a suitably large number. In addition, one could also consider the cost of doing an action in a given state given by a function $C: S \times A \rightarrow [0, MaxCost]$, where *MaxCost* is the maximum possible cost of doing an action. We can now define the *utility* of being in a state $s \in S$ and applying an action $a \in A$ as,

$$U(s, a) = R(s) - C(s, a)$$

If we are considering a policy π then the utility of applying the policy in state $s \in S$ is given by,

$$U(s|\pi) = R(s) - C(s, \pi(s))$$

where $a = \pi(s)$ is the action chosen by the policy in that state.

Given the notion of utility of being in a state, one can define a *value function* of a state given a policy p as a function of the total reward, accumulated over the history $h = \langle s_0, s_1, \dots \rangle$ minus the total cost involved, when the policy is applied. That is,

$$V(s_0|\pi) = u(U(s_0|\pi(s_0)), U(s_1|\pi(s_1)), \dots)$$

where u is some utility accumulating function. In the general case when the system may run for ever, one would like to measure the “current worth” of a state using a function that gives lesser importance to states that occur farther into the future. This is akin to discounting the future value of money now. It is not surprising that the idea of measuring current worth by discounting the future value has been borrowed from economics. In fact, MDPs were first devised in economics. A common utility accruing function sums up the individual utilities multiplied by a value based on a *discounting factor* $0 \leq \gamma \leq 1$.

$$V(s_0|\pi) = \sum_{t \geq 0} \gamma^t (R(s_t) - C(s_t, \pi(s_t)))$$

The smaller the value of γ , the greater the importance given to states nearer to the time $t = 0$. The sequence of states over which the value is computed is known as a history $h = \langle s_0, s_1, s_2, \dots \rangle$. The above equation says that for any history starting with the state s_0 and going through states s_1, s_2, \dots , and so on, results in a value that can be assigned as the value of the initial state. What is the role of the policy here? The policy is the one that decides what actions are taken in each state, and therefore

what state the system potentially moves to. However, given a policy π , one could have several histories generated from the starting state s_0 , and in fact several other histories that could be generated starting at other states. The utility of a policy can then be defined as the expected value accrued over all such histories, multiplied by the probability of the history occurring given π . Let H be the set of all possible histories and let $h \in H$ be some history and let s_h be the starting point of that history h (Ghallab et al., 2004).

$$E(\pi) = \sum_{h \in H} P(h|\pi) V(s_h|\pi)$$

The task of planning with MDPs is to find an *optimal policy* π^* , such that the expected utility of the policy π^* is greater than the expected utility of any other policy π , that is, $E(\pi^*) > E(\pi)$.

One can distinguish three kinds of MDPs. The first are the *finite horizon* MDPs in which there is a bound on the number of actions that can be executed. The second, *infinite horizon* MDPs with discounted costs and rewards. The idea here is that a discounting factor $0 \leq \gamma \leq 1$ gives lower weight to costs and rewards in the future. The lower is the value of γ , the lesser the importance given to the (distant) future. The third class of MDPs are *indefinite horizon* MDPs in which the task is expected to be completed in a finite time, but with uncertainty in the time duration (like an aeroplane circling over an airport to get landing clearance, or a girl attempting to shoot a basket on a basketball court). It turns out that all three kinds of MDPs can be generalized into a stochastic shortest path MDP (SSP MDP) described below (Mausam and Kolobov, 2012).

Stochastic shortest path MDPs introduce the notion of goal states. In that sense, they are similar to classical planning problems. However, they have a notion of costs that can incorporate the notion of rewards used in MDPs. By changing the sign of a positive reward, it can be incorporated into the cost function. In this way, SSP MDPs can allow for preferences and trajectory constraints as well. An SSP MDP is defined as a tuple $\langle S, A, P, C, G \rangle$ where,

- S is a finite set of states.
- A is a finite set of actions.
- $P: S \times A \times S \rightarrow [0,1]$ is the state transition probability $P(s, a, s')$ of going from state s to state s' by action a applied is state s .
- $C: S \times A \times S \rightarrow [0,\infty)$ is the nonzero cost $C(s, a, s') > 0$ of going from state s to state s' by action a applied is state s . The cost of going to a *goal state* from the goal state is however 0.
- $G \subseteq S$ is a set of goal states which satisfy the following properties.
For all $s_g \in G$, for all $a \in A$ and for all $s' \notin G$,

$P(s_g, a, s') = 0$, no action can take the system away from the goal state,

$P(s_g, a, s_g) = 1$, any action in the goal state keeps it in the goal

state,

$C(s_g, a, s_g) = 0$, the cost of staying in the goal state is 0.

There is a condition that the SSP³⁸ should have a *proper policy*. A policy is called proper, if from any state $s \in S$ the policy π will drive it to a goal state in a finite amount of time. In other words, the system cannot loop amongst nongoal states indefinitely.

Dealing with SSPs, the optimality criterion now becomes the *minimization* of the total expected cost, as opposed to maximization of rewards collected. The discussion which follows applies to SSPs. The reader should however keep in mind that other MDPs generalize to SSPs, as a consequence the conclusions apply to all MDPs.

We are interested in choosing between competing policies. A policy specifies an action in each state. The underlying state transition system specifies the actions that are possible in each state. Consider the following example state space depicted in Figure 17.29 which is an SSP MDP. One could think of it as climbing a slippery snow slope from point s_1 to point s_g . There are two possible routes, one via s_2 and the other via s_5 . However, each action is fraught with the danger of slipping, shown by two arrows (state transitions) emerging from the same point in each node. For example, attempting to go from s_1 to s_5 has the danger that one might slip and end up in s_4 instead, from where the only option is to go back to s_1 (and try again).

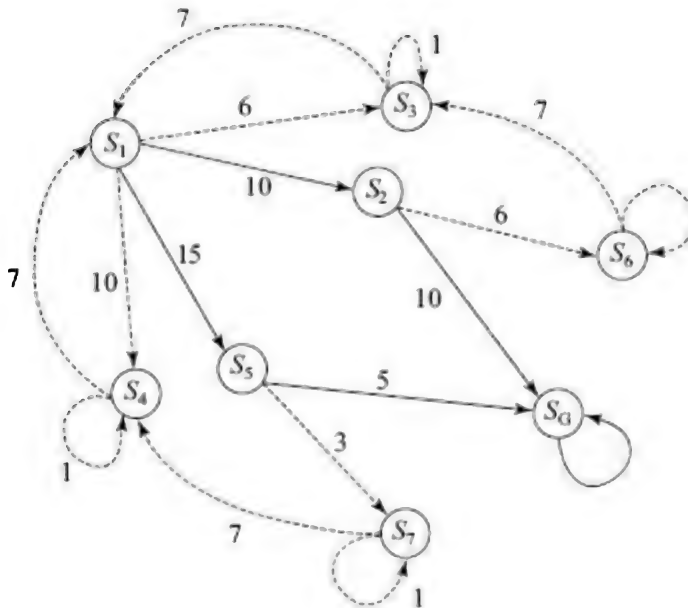


Figure 17.29 A stochastic planning domain. The two arrows originating from the same point on the nodes represent the two possible state transitions given an action. The four arrows with solid lines represent the desired paths. The labels on the edges are costs of actions.

The action set available to the agent is $A = \{a_{12}, a_{13}, a_{14}, a_{15}, a_{2G}, a_{26}, a_{5G}, a_{57}, a_{41}, a_{74}, a_{63}, a_{31}\}$. The stochastic behaviour of the actions is described in the Table 17.13. In addition, we assume that each state s_k has an action called *stay* that can be executed deterministically. The effect of the *stay* action is to remain in the same state.

Table 17.13 The probabilistic state transitions for the given actions

Action	Source	Destination (probability)	
a_{12}	s_1	s_2 (0.8)	s_3 (0.2)
a_{13}	s_1	s_2 (0.0)	s_3 (1.0)
a_{14}	s_1	s_4 (0.6)	s_5 (0.4)
a_{15}	s_1	s_5 (0.9)	s_4 (0.1)
a_{2G}	s_2	s_G (0.7)	s_6 (0.3)
a_{26}	s_2	s_G (0.0)	s_6 (1.0)
a_{5G}	s_5	s_G (0.6)	s_7 (0.4)
a_{57}	s_5	s_G (0.0)	s_7 (1.0)
a_{31}	s_3	s_1 (0.6)	s_3 (0.4)
a_{41}	s_4	s_1 (0.6)	s_4 (0.4)
a_{63}	s_6	s_3 (0.7)	s_6 (0.3)
a_{74}	s_7	s_4 (0.7)	s_7 (0.3)

Given the above problem, one can immediately think of two interesting policies, one trying to reach s_G via s_2 and the other trying to reach s_G via s_5 . The policies π_{2G} and π_{5G} are described below in which the action in each state $s \in S$ is prescribed.

The actions in each of the eight states as per π_{2G} are,

$$\pi_{2G}(s_1) = a_{12}$$

$$\pi_{2G}(s_2) = a_{2G}$$

$$\pi_{2G}(s_3) = a_{31}$$

$$\pi_{2G}(s_4) = a_{41}$$

$$\pi_{2G}(s_5) = a_{5G}$$

$$\pi_{2G}(s_G) = \text{stay}$$

$$\pi_{2G}(s_6) = a_{63}$$

$$\pi_{2G}(s_7) = a_{74}$$

Observe that the policy directs the agent to take actions a_{12} whenever it is in state s_1 . The only unintended effect of this action is to take the agent to s_3 instead of s_2 , from where the only moving action is back to s_1 . This means that, given the above set of action effects, there is no chance of the agent landing up in state s_5 . However, the policy does specify the action to be taken in that state.

The other policy of interest p_{5G} has the following actions,

$$\pi_{5G}(s_1) = a_{15}$$

$$\pi_{5G}(s_2) = a_{2G}$$

$$\pi_{5G}(s_3) = a_{31}$$

$$\pi_{5G}(s_4) = a_{41}$$

$$\pi_{5G}(s_5) = a_{5G}$$

$$\pi_{5G}(s_G) = \text{stay}$$

$$\pi_{5G}(s_6) = a_{63}$$

$$\pi_{5G}(s_7) = a_{74}$$

The reader would have noticed that the only place this policy differs from the previous one is in the action to be taken in state s_1 . In all other states, the actions are identical. This is not surprising, given that these are the only two that are likely to be the desired policies. The reader is encouraged to verify that all other policies (for example the one with $\pi_{2G}(s_1) = a_{13}$) are not going to work.

The other policies are not likely to be of interest. Nevertheless, the above arguments were made at a qualitative level. We need a quantitative approach that will choose the “best” policy from the set of all possible policies. This will also involve choosing between one of π_{2G} and π_{5G} amongst others.

Given the utility function, one can then look at a policy that maximizes the overall expected utility obtained by applying the policy. The first step however is to evaluate a given policy.

How to Evaluate a Policy?

If one is to look at different policies in search of the optimum policy, one needs a mechanism to evaluate any policy. A policy can be depicted by a policy hyper-graph in which exactly one directed hyper-edge, representing the prescribed action, emanates from each state, and ends in all the states that the action could end up in.

Consider first a policy that has no cycles. Consider a simpler version of the problem from Figure 17.29 shown in Figure 17.30 in which there are only four states, s_1 , s_2 , s_5 and s_G where s_G , as before, is the goal state. We have removed some states from the original problem to allow us to select a *proper policy*. A proper policy is one in which an agent is guaranteed to reach a goal state. If we had left states s_3 , s_4 , s_6 and s_7 in, and removed the backward moves, then these states could have been *dead-end* states. An SSP with dead-end states requires greater sophistication because in addition to the cost that one has to minimize, one will also have to take into account the probability of reaching a goal state, and perhaps a trade off between the two criteria.

Let agent choose a policy π for the *finite horizon* problem consisting of the following actions,

$$\pi(s_1) = a_1$$

$$\pi(s_2) = a_2$$

$$\pi(s_5) = a_5$$

$$\pi(s_G) = \text{stay}$$

The corresponding policy graph along with probabilities and costs associated with the prescribed actions is shown in Figure 17.30. Observe that only action a_1 is stochastic, and the probability of reaching the goal in two steps is 1.

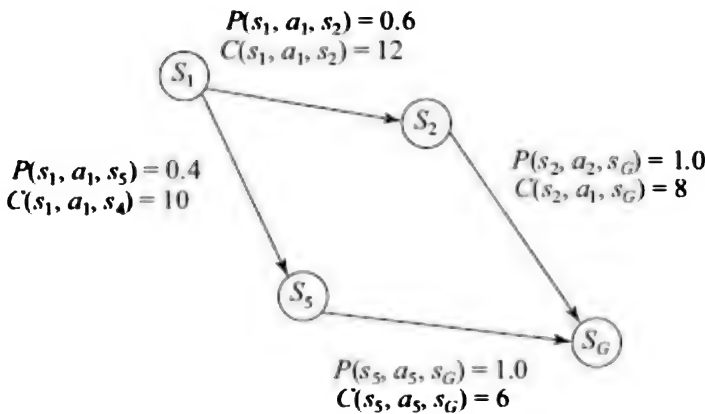


Figure 17.30 A policy graph for finite domain MDP problem. The first action is nondeterministic leading to one of s_2 or s_5 , from where the two chosen actions are deterministic.

We would now like to compute the *value* of each state, which is the total expected cost of reaching a goal state starting in that state. Let us call this function as V^π the valuation function given the policy π .

$$\begin{aligned}
V^\pi(s_G) &= V(s_G|\pi) = 0 \\
V^\pi(s_5) &= V(s_5|\pi) = C(s_5, a_5, s_G) = 10 \\
V^\pi(s_2) &= V(s_2|\pi) = C(s_2, a_2, s_G) = 8 \\
V^\pi(s_1) &= V(s_1|\pi) = P(s_1, a_1, s_2) [C(s_1, a_1, s_2) + C(s_2, a_2, s_G)] \\
&\quad + P(s_1, a_1, s_5) [C(s_1, a_1, s_5) + C(s_5, a_5, s_G)] \\
&= 0.6 [12 + 8] + 0.4 [10 + 6] \\
&= 0.6 \times 20 + 0.4 \times 16 \\
&= 18.4
\end{aligned}$$

The state s_G has value zero because it is already the goal state. The value of s_5 is 6 because a_5 is a deterministic action leading to the goal state with cost 6. Similarly, the cost of s_2 is 8. The value of s_1 is more involved. There are two ways of reaching the goal. One via s_2 has total cost 20 and has a probability of 0.6 being taken. The other via s_5 has cost 16 and has probability 0.4 being taken. The expected cost is then the sum of the costs of the two routes multiplied by the probability of each route being taken.

Very often however, policies are cyclic. That is, one may revisit the same state again. In fact, it would be quite likely that in a stochastic domain, an agent may need to try some actions repeatedly till it succeeds³⁹. The values of such cyclic policies can be computed by setting up a system of linear equations as follows. The long term cost reaching a goal from a given state s is the cost of making the first move, plus the cost of reaching the goal from the destination of the first move. Since the action may be stochastic, the first move may end up in different states with different probabilities, and a weighted average would have to be taken.

$$\begin{aligned}
V^\pi(s) &= 0 && \text{if } s \in G \\
&= \sum_{s' \in S} P(s, a, s') [C(s, a, s') + V^\pi(s')] && \text{otherwise}
\end{aligned}$$

Let us write down these equations for the policy π_{5G} for the problem depicted in Figure 17.29. The policy graph is depicted in Figure 17.31. Observe that paths from all states *eventually* lead to the goal state s_G .

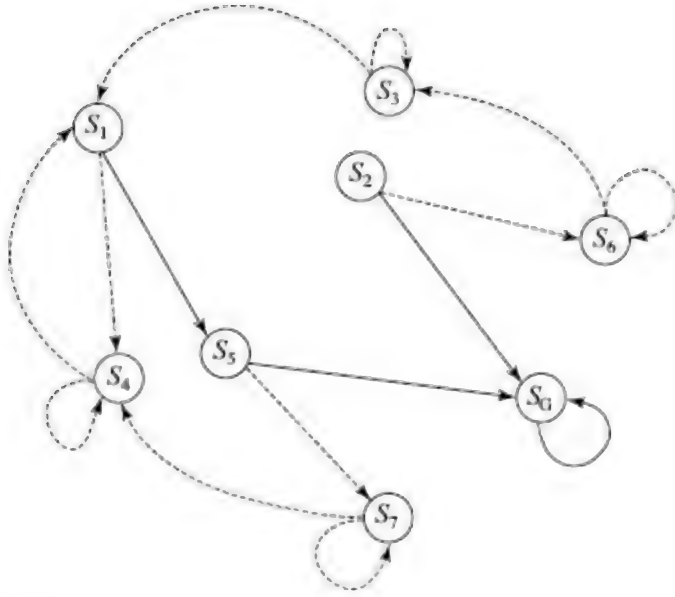


Figure 17.31 The policy graph for the cyclic policy π_{5G} .

The equations are given below. We have written the expression only for the first one.

$$\begin{aligned}
 V^{\pi_{5G}}(s_1) &= P(s_1, a_{15}, s_3)[C(s_1, a_{15}, s_3) + V^{\pi_{5G}}(s_3)] + P(s_1, a_{15}, s_4)[C(s_1, a_{15}, s_4) + V^{\pi_{5G}}(s_4)] \\
 V^{\pi_{5G}}(s_1) &= 0.9[15 + V^{\pi_{5G}}(s_3)] + 0.1[10 + V^{\pi_{5G}}(s_4)] \\
 V^{\pi_{5G}}(s_2) &= 0.7[10 + V^{\pi_{5G}}(s_G)] + 0.3[6 + V^{\pi_{5G}}(s_6)] \\
 V^{\pi_{5G}}(s_3) &= 0.6[7 + V^{\pi_{5G}}(s_1)] + 0.4[1 + V^{\pi_{5G}}(s_3)] \\
 V^{\pi_{5G}}(s_4) &= 0.6[7 + V^{\pi_{5G}}(s_1)] + 0.4[1 + V^{\pi_{5G}}(s_4)] \\
 V^{\pi_{5G}}(s_5) &= 0.6[5 + V^{\pi_{5G}}(s_G)] + 0.4[3 + V^{\pi_{5G}}(s_7)] \\
 V^{\pi_{5G}}(s_6) &= 0.7[7 + V^{\pi_{5G}}(s_3)] + 0.3[1 + V^{\pi_{5G}}(s_6)] \\
 V^{\pi_{5G}}(s_7) &= 0.6[7 + V^{\pi_{5G}}(s_4)] + 0.3[1 + V^{\pi_{5G}}(s_7)] \\
 V^{\pi_{5G}}(s_G) &= 0
 \end{aligned}$$

Solving these equations we get the values listed for $[V^{\pi_{5G}}(s_1), \dots, V^{\pi_{5G}}(s_7)]$ as,

$$V^{\pi_{5G}} = [45.3351, 26.9291, 53.0018, 53.0018, 28.3721, 60.4303, 60.4303]$$

One can see that the expected costs are pretty high compared to the costs of deterministic solutions. The highest costs are for $V^{\pi_{5G}}(s_6)$ and $V^{\pi_{5G}}(s_7)$, which are symmetrically placed in the state space. They have to get all the way back to s_1 and try going through s_5 again. The expected costs are lowest for s_2 and s_5 , since they are closest to the goal node and the policy picks the action taking them to the goal node.

The policy π_{2G} is the other interesting policy (that could be optimal). The reader should also verify that any other policy is going to be more expensive. For π_{2G} on the action at state s_1 is different and so the

corresponding equation is,

$$\begin{aligned} V^{\pi^2G}(s_1) &= P(s_1, a_{12}, s_2)[C(s_1, a_{12}, s_2) + V^{\pi^2G}(s_2)] + P(s_1, a_{12}, s_3)[C(s_1, a_{12}, s_3) + V^{\pi^2G}(s_3)] \\ &= 0.8[10 + V^{\pi^2G}(s_2)] + 0.2[6 + V^{\pi^2G}(s_3)] \end{aligned}$$

The rest of the equations are the same and the values for $[V^{\pi^2G}(s_1), \dots, V^{\pi^2G}(s_7)]$ are,

$$V^{\pi^2G} = [38.2075, 24.7908, 45.8741, 45.8741, 25.5211, 53.3027, 53.3027]$$

One can see that the lower costs of actions a_{12} and a_{2G} are reflected in the overall expected costs.

The costs are higher than the deterministic costs because the probabilities of heading away from the path to the goal are significant. Let us modify the probabilities a little to get a sense of how these probabilities influence expected costs. Let us change the probabilities of the actions from s_4 and s_7 to much higher probabilities for heading towards s_1 , as opposed to staying put.

Table 17.14 The modified probabilities for a_{14} and a_{74}

a41	s_4	s_1 (0.95)	s_4 (0.05)
a74	s_7	s_4 (0.95)	s_7 (0.05)

The corresponding changed equations for the two states s_4 and s_7 for which these two actions are prescribed by both policies are,

$$\begin{aligned} V^{\pi^5G}(s_4) &= 0.95[7 + V^{\pi^5G}(s_1)] + 0.05[1 + V^{\pi^5G}(s_4)] \\ V^{\pi^5G}(s_7) &= 0.95[7 + V^{\pi^5G}(s_4)] + 0.05[1 + V^{\pi^5G}(s_7)] \end{aligned}$$

Using these equations, along with the others for π_{5G} gives us the values,

$$V^{\pi^5G} = [44.812, 26.7722, 52.4787, 51.8647, 27.9173, 59.9073, 59.2932]$$

And for π_{2G} we get the values,

$$V^{\pi^5G} = [38.2075, 24.7908, 45.8741, 45.8741, 25.3707, 53.3027, 52.9268]$$

One can observe that increasing the chances of getting back to square one quickly does not do much for the expected costs. This is because the *costs incurred* in staying put at s_4 and s_7 are very low, and therefore the savings by avoiding them are low as well.

Instead, if one were to increase the probability of action a_{5G} succeeding by making $P(s_5, a_{5G}, s_G) = 0.9$, we get the equation,

$$V^{\pi^5G}(s_5) = 0.9[5 + V^{\pi^5G}(s_G)] + 0.1[3 + V^{\pi^5G}(s_7)]$$

Replacing this in the set of linear equations and solving them we get the values for π_{5G} as,

$$V^{\pi^5G} = [25.8583, 21.0861, 33.525, 33.525, 8.89536, 40.9536,$$

40.9536]

As expected, the increased probability of going from s_5 to s_G has reduced the chance of the system looping back to s_1 and accumulating more cost. In fact, the expected cost of going from s_1 to s_G is now 25.86, which is much closer to the ideal cost of 20.

On the other hand, increasing the probability of going from s_5 to s_G has not affected the expected costs for policy π_{2G} because the policy drives the solution through s_2 . The only state for which the value has gone down is s_5 , which is the cost if you happen to start from there. Observe that even this is a little higher than the cost from s_5 in π_{5G} , because when the action a_{5G} does not result in reaching s_G , which happens 10% of the times, the agent has to go back to s_1 and find a path via s_2 .

$$V^{\pi_{2G}} = [38.2075, 24.7908, 45.8741, 45.8741, 10.1303, 53.3027, 53.3027]$$

Solving a set of equations with N variables can be quite expensive. Instead, one can also adopt an iterative approach to evaluating a policy.

Iterative Policy Evaluation

The iterative policy evaluation algorithm initializes the values of all variables and then iterates through the linear equations by computing the new left hand sides using the old values in the right hand side. Let V^n stand for the value function in the n^{th} iteration. The values for each state variable that is not a goal state are then updated as,

$$V^{\pi}_{n+1}(s) = \sum_{s' \in S} P(s, a, s') [C(s, a, s') + V^{\pi}_n(s')]$$

Applying this iterative process, the sets of equations for the policy π_{5G} , we have the following iterations in which $V^{\pi_{5G}}_n$ is replaced with 0 for all n .

$$\begin{aligned} V^{\pi_{5G}}_{n+1}(s_1) &\leftarrow 0.9[15 + V^{\pi_{5G}}_n(s_5)] + 0.1[10 + V^{\pi_{5G}}_n(s_4)] \\ V^{\pi_{5G}}_{n+1}(s_2) &\leftarrow 0.7[10 + V^{\pi_{5G}}_n(s_G)] + 0.3[6 + V^{\pi_{5G}}_n(s_6)] \\ V^{\pi_{5G}}_{n+1}(s_3) &\leftarrow 0.6[7 + V^{\pi_{5G}}_n(s_1)] + 0.4[1 + V^{\pi_{5G}}_n(s_3)] \\ V^{\pi_{5G}}_{n+1}(s_4) &\leftarrow 0.6[7 + V^{\pi_{5G}}_n(s_1)] + 0.4[1 + V^{\pi_{5G}}_n(s_4)] \\ V^{\pi_{5G}}_{n+1}(s_5) &\leftarrow 0.6[5 + V^{\pi_{5G}}_n(s_G)] + 0.4[3 + V^{\pi_{5G}}_n(s_7)] \\ V^{\pi_{5G}}_{n+1}(s_6) &\leftarrow 0.7[7 + V^{\pi_{5G}}_n(s_3)] + 0.3[1 + V^{\pi_{5G}}_n(s_6)] \\ V^{\pi_{5G}}_{n+1}(s_7) &\leftarrow 0.6[7 + V^{\pi_{5G}}_n(s_4)] + 0.3[1 + V^{\pi_{5G}}_n(s_7)] \end{aligned}$$

There are several questions one can ask of this iterative process. The following observations have been made in (Mausam and Lolobov, 2012),

- Given a proper policy, the algorithm converges to a fixed point which is both unique and optimal. The values it converges to are the solutions to the set of linear equations the iterative process is based upon.
- The running time of the algorithm can be controlled by using a threshold ϵ on the *residual*. The residual is the magnitude of the difference in the value for successive iterations, that is $|V^{\pi}_{n+1}(s) - V^{\pi}_n(s)|$. By choosing a threshold ϵ appropriately, the values produced can be made ϵ -consistent.
- Each update of a variable, in the general case will require using all the neighbours in the state graph requiring $O(|S|)$ time, where $|S|$ is the size of the state space. Since updates have to be done for each state, one iteration needs $O(|S|^2)$ time.

The graph in Figure 17.32 shows the iterative values for the policy π_{5G} starting with a value of 0 at $n = 1$ for every state. The graph also plots the residual for state s_1 .

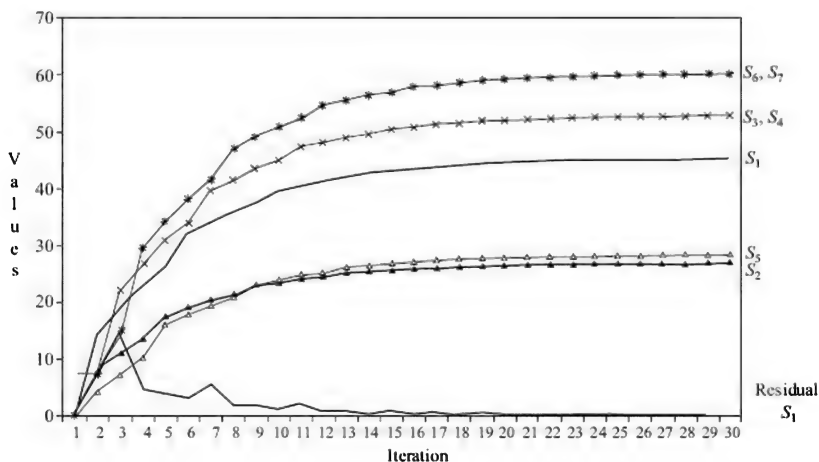


Figure 17.32 A plot of iterated value functions starting with the value 0 for all states. The bottom of the graph plots the residual for the value of state s_1 .

The value of the residual for s_1 drops to 0.153651 in the 20th iteration. This could be a reasonable point to stop the process. The values after 29 iterations are,

$$V^{\pi_{5G}} = [45.2375, 26.8677, 52.8811, 52.8811, 28.2903, 60.2627, 0.0368]$$

which are close to the values computed by solving the equations. Figure 17.33 shows the evolution of three states s_1 , s_5 and s_6 , starting from two sets of initial values for all the seven variables, 0 and 100. The plot shows that irrespective of the initial values, the algorithm converges to the same set of values.

17.6.2 Solving MDPs

The task of planning with MDPs is to find the optimal policy for a given problem statement. Recall that the problem can be generalized to SSP in which the task is to minimize the overall expected costs of reaching a goal state. A brute force approach would evaluate all possible policies and pick the best one. However, the number of different policies can be very large. If there are N states in the system and a choice of K actions in each state, one would need to evaluate K^N policies each needing an order of N^2 computations.

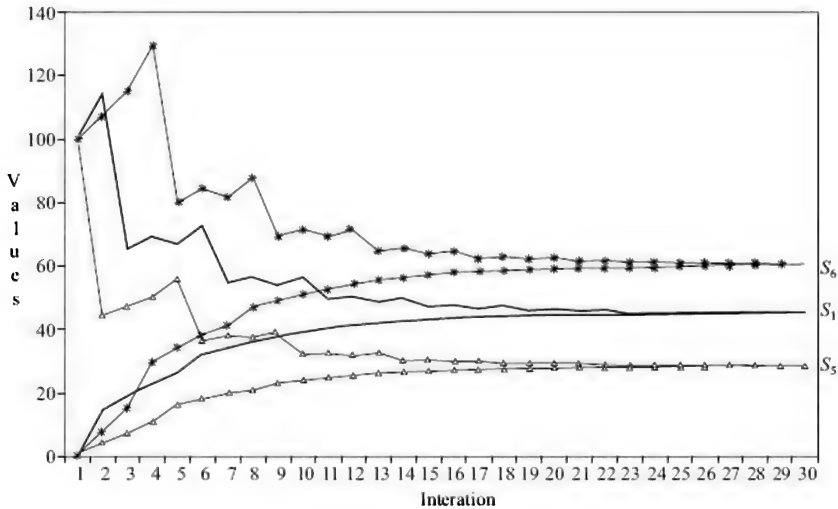


Figure 17.33 The values of states S_1 , S_5 and S_6 starting with values 0 and 100 converge to the same values for the two initial values.

There have been two approaches to finding optimal policies faster, policy iteration and value iteration. We look at them below.

Policy Iteration

Given that we can evaluate a policy, how does one search for an optimal policy. There are two issues here. One, how does one compare two policies, and two, how does one pick the best policy, given that you can compare two policies. The definition of an optimal policy says that the expected cost over all possible histories must be optimal. Given two policies π_1 and π_2 we say that policy π_1 is better than π_2 if,

$$\forall s \in S (V^{\pi_1}(s) \geq V^{\pi_2}(s))$$

The basic idea behind *Policy Iteration* is to start with a random policy and find a better policy in each iteration. Let π_0 be the initial policy. Then,

in each iteration, the algorithm moves from a policy π_{n-1} to a new policy π_n which has a better action prescribed for some state. The quality of an action a in a state s is measured by a factor known as its Q-value defined as (Howard, 1960),

$$Q^V(s, a) = \sum_{s' \in S} P(s, a, s') [C(s, a, s') + V(s')]$$

where $V(s)$ is assumed to be the true expected cost of reaching the goal from state s . The Q-value of a state is the expected value of the state when a given action is taken in that state.

Since one does not have access to the true expected cost, one goes through an iterative process which oscillates between two phases. In the *policy evaluation* phase, the algorithm evaluates the value function for a *given* policy. This can be done by solving the corresponding set of linear equations or an iterative procedure as described in the previous section. In the *policy improvement* phase, the current policy is refined to a new policy with a smaller value function. This is done by a process called *greedy policy construction* which is as follows. Given a value function V , a greedy policy π^V selects a locally optimal action at each state by a process of one step look ahead. That is,

$$\begin{aligned} \pi^V(s) &= \operatorname{argmin}_{a \in A} Q^V(s, a) \\ &= \operatorname{argmin}_{a \in A} \sum_{s' \in S} P(s, a, s') [C(s, a, s') + V(s')] \end{aligned}$$

That is, it selects an action that yields the lowest expected cost from state s . If, and only if, the new value of the state s becomes lower than the old value, then the action is incorporated into the policy. In this way, it moves to a new policy which has a strictly lower cost. The process continues till a better policy cannot be constructed. The algorithm in the Figure 17.34 below is adapted from (Mausam and Kolobov, 2012). We use the notation Q_{n-1}^V to represent the Q-value at the end of the $(n-1)^{\text{th}}$ iteration generated by the policy π_{n-1} . The expression V_{n-1}^π represents the complete value function and the end of the $(n-1)^{\text{th}}$ iteration, and $V_n(s_k)$ is the lowest value for state s_k amongst the value generated by all possible actions applicable to the state in the current iteration. As can be seen from lines 11–12, when this value is lower than the earlier value in which action $\pi_{n-1}(s_k)$ was applied then the action is replaced by the action that yields the lowest value.

```

Policy-Iteration (state space  $S$ , action set  $A$ , costs  $C$ )
1  $n \leftarrow 0$ 
2 for  $k \leftarrow 1$  to  $|S|$ 
3   do  $\pi_0(s_k) \leftarrow a_{k1}$  /* initialize the policy randomly */
4 repeat
5    $n \leftarrow n + 1$ 
6   Compute  $V_{n-1}^*$  /*Solve the set of linear equations */
7   for  $k \leftarrow 1$  to  $|S|$ 
8     do  $\pi_n(s_k) \leftarrow \pi_{n-1}(s_k)$ 
9        $\forall a \in A$  Compute  $Q_{n-1}^V(s_k, a)$ 
10       $V_n(s_k) \leftarrow \min_{a \in A} Q_{n-1}^V(s_k, a)$ 
11      if  $Q_{n-1}^V(s_k, \pi_{n-1}(s_k)) > V_n(s_k)$ 
12        then  $\pi_n(s_k) \leftarrow \operatorname{argmin}_{a \in A} Q_{n-1}^V(s_k, a)$ 
13 until  $\pi_n = \pi_{n-1}$ 
14 return  $\pi_n$ 

```

Figure 17.34 The algorithm *Policy Iteration* starts with a random policy, in this version, choosing the first applicable action in each state. It then goes through a loop looking for better actions for each state s_k . An action is better than the previous one, if it results in a lower Q -value for that state.

The point to note is that the *Policy Iteration* algorithm refines policies to strictly better policies. In each cycle of the *Repeat* loop, the policy is guaranteed to improve. It is doing *Hill Climbing* in the policy state (see Chapter 3). However, there is no danger of getting stuck in a local minimum. It has been proved that if the algorithm is initialized with a proper policy π_0 then it is guaranteed to find the optimal policy.

One has to be careful that the initial policy is a proper one, if one is to use *Policy Iteration*. The next algorithm we look at is free from any such problem.

Value Iteration

The *Value Iteration* algorithm devised by Richard Bellman in (1957) focuses directly on the value functions and searches in the value function space. The Bellman equations (also known as dynamic programming equations) given below, capture the optimality criteria for solving an MDP, which is an example of dynamic programming.

$$Q^*(s,a) = \sum_{s' \in S} P(s', a, s) [C(s, a, s') + V^*(s')]$$

where $Q^*(s,a)$ is the expected cost of executing the action a in state s and then following the *optimal* policy, yielding the optimal value $V^*(s')$ for the resulting state. The optimal policy in turn is defined as one that chooses the action that optimizes the Q -value,

$$\begin{aligned}
 V^*(s) &= 0 && \text{if } s \in G \\
 &= \min_{a \in A} Q^*(s,a) && \text{otherwise}
 \end{aligned}$$

The Bellman equations have a unique solution that corresponds to the value of the optimal policy π^* . The first equation specifies how to

compute the Q-value, given the optimal values $V^*(S')$ of all successor states, and the second one specifies that the optimal value $V^*(s)$ for a state is obtained by choosing the action in that state that yields the lowest Q-value. The Bellman equations were first applied to engineering control theory and to other topics in applied mathematics, and subsequently became an important tool in economic theory, before being adopted by the probabilistic planning community.⁴⁰

The *Value Iteration* algorithm begins by initializing the value function to some random value. Let us call the initial value function V_0 . Then it goes through an iterative refinement process till the consecutive values become ϵ -consistent. In each iteration, it sweeps over the entire state space updating the value function V_n in the n^{th} iteration, based on the value function V_{n-1} . The update procedure is known as the *Bellman update* or the *Bellman backup*.

$$V_{n+1}(s) \leftarrow \min_{a \in A} \sum_{s' \in S} P(s, a, s') [C(s, a, s') + V_n(s')]$$

The reader is encouraged to compare the Bellman backup rule with the iterative policy evaluation rule (Section 17.6.1.2). The main difference is that there a policy is specified and its value has to be evaluated; while in the Bellman backup, one does not have access to a policy and has to search for both the optimal policy and the corresponding, optimal value function simultaneously.

The *Value Iteration* procedure ends with the optimal value function V^* . Given the optimal value function, the optimal policy π^* can be constructed by the greedy approach that constructs π^V given V , described in the preceding section.

The algorithm adapted from (Mausam and Kolobov, 2012) is described in Figure 17.35. First, the policy function is initialized randomly. Then, the Bellman backup is applied iteratively till the maximum residual for any state becomes less than a specified value ϵ (lines 4–9). The algorithm returns the optimal policy constructed by the greedy approach described above.

```

Value-Iteration (state space  $S$ , action set  $A$ , costs  $C$ ,  $\epsilon$ )
1   $n \leftarrow 0$ 
2  for  $k \leftarrow 1$  to  $|S|$ 
3      do  $V_0(s_k) \leftarrow \text{random-value}$  /* initialize the values randomly */
4  repeat
5       $n \leftarrow n + 1$ 
6      for  $k \leftarrow 1$  to  $|S|$ 
7          do Compute  $V_n(s_k)$  using Bellman backup
8              Compute  $\text{Residual}_n(s_k) = |V_n(s_k) - V_{n-1}(s_k)|$ 
9  until  $\max_{s' \in S} \text{Residual}_n(s) < \epsilon$ 
10 return  $\forall s \in S \pi^V(s) = \operatorname{argmin}_{a \in A} \sum_{s' \in S} P(s, a, s') [C(s, a, s') + V(s')]$ 

```

Figure 17.35 The *Value Iteration* algorithm begins by initializing the value function for each state randomly. Then it applies the Bellman update to each state till all states have reached the desired threshold for the residual. It then constructs and returns the greedy policy π^V for the value function.

As discussed above, the *Value Iteration* algorithm is similar to the

iteration process used for policy evaluation. However, the *Value Iteration* algorithm achieves much more. It explores the policy space, choosing the action that yields the lowest Q-value at each step, since it does not have the benefit of access to a policy.

Figure 17.36 shows the progress of the algorithm for the SSP problem depicted in Table 17.13. As the plot shows, the residual for the state s_1 has dropped to zero by the 31st iteration. The reader should observe that the value function is close to the value function obtained by solving the linear equations for policy π_{2G} , reproduced below, and also shown in the figure. Our informal discussion had suggested that π_{2G} , the better of the two policies we looked at, is the optimal policy, and that has been borne out by the *Value Iteration* algorithm.

The values [$V^{p_{2G}}(s_1), \dots, V^{p_{2G}}(s_7)$] computed by solving the linear equations,
 $V^{p_{2G}} = [38.2075, 24.7908, 45.8741, 45.8741, 25.5211, 53.3027, 53.3027]$

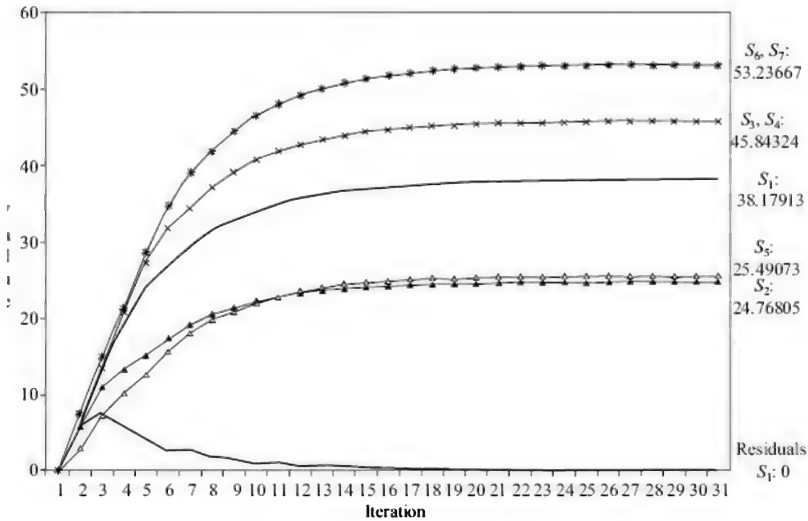


Figure 17.36 The progress of the Value Iteration starting with all states initialized to 0. After 31 iterations, the residual of S_1 becomes 0. Observe that the Value function approaches the Value function of policy π_{2G} which is the optimal policy.

The plot in Figure 17.36 has been obtained by initializing all values of the value function to 0 for the sake of illustration. In practice, one would use more informed initial values. For example, one could use the costs of reaching the goal for deterministic actions. The above plot appears similar to the plot in Figure 17.32, but one must keep in mind that the *Value Iteration* algorithm has to consider all possible actions at each step, as compared to a prescribed action in evaluating a given policy.

The *Value Iteration* algorithm converges to the optimal value functions for any initial values. This is an advantage over the *Policy Iteration*

algorithm that required the initial policy to be proper.

Application of each instance of the Bellman update may access all states and all actions, and is therefore of order $|S| \times |A|$. The backup is allied to each state and therefore the computational complexity of each iteration is $O(|S|^2 \times |A|)$.

Attempts to improve the computational complexity of *Value Iteration* have been along the following lines. The Gauss-Seidel version makes the updated values of states available for other states in the *same* iteration. Asynchronous *Value Iteration* algorithms question the need to update each and every state in each iteration. Prioritization of the order in which states are selected is another approach. The idea is to select those states for updating earlier which are likely to change. For example, if the successors of a state in the transition graph have not changed, then backing up values from them is not going to change the value of the state. The interested reader is encouraged to refer to (Mausam and Kolobov, 2012) for a detailed discussion of these algorithms.

In the following section, we look at an algorithm that employs *heuristic search* to cut down on the states that need to be explored.

Lao*

Computing a policy is a little bit like implementing Dijkstra's shortest path algorithm for a complete graph, in the sense that a policy specifies the optimal actions for all possible starting states. Given our focus on planning, algorithms that explore the state space for reaching a goal state from a given start state, are of interest.

We look at an algorithm *LAO** (Hansen and Zilberstein, 2001) that generalizes the heuristic search algorithm *A** (see Chapter 5) and *AO** (see Chapter 6) to solve planning problems formalized as MDPs. The main idea of heuristic search is to use a heuristic function that *estimates* the distance or cost to the goal state and drives the search towards those states that *seem* more promising. As in *A** and *AO**, if one employs a heuristic function that underestimates the actual optimal cost, even in *LAO** one is guaranteed to find the optimal solution without necessarily exploring the entire state space.

The transition graph G_S of the domain can be viewed as an AND-OR (AO) graph. A stochastic action from any state s is connected by a *hyper-edge* to a set of nodes, to *one of* which the system will move when the action is taken. Such actions are depicted by AND arcs or directed hyper-edges emanating from a node and ending up in the set of nodes that the action can lead to. They are also known as *k-connectors*. Figure 17.37 shows the stochastic domain of Figure 17.29 as an AO graph. Each hyper-edge is formed by arrows linked together by arcs. The stay actions have not been depicted, except for the goal state s_G where it is the only action.

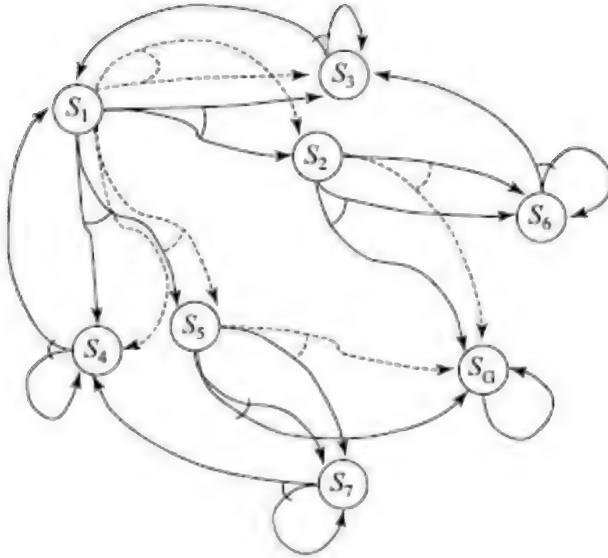


Figure 17.37 A stochastic planning domain can be seen as an AND-OR graph. Each stochastic action is seen as an AND edge, leading to (one of) a set of states. The AND edge is a hyper-edge, depicted by a set of edges are connected together by an arc. The different actions available at each state are OR choices. Some actions have been drawn with dashed lines for clarity.

The *scope of exploration* for the LAO* algorithm is the subgraph G_{Start} of the transition rooted at a given start state $Start$. Even for G_{Start} , the algorithm may explore only a small part depending upon how good the heuristic function it uses is. In our example, we will assume that the start node is $s1$ in the figure below. Observe that in this case, $G_{Start}=G_{s1}$ is the entire transition graph.

The solution for a deterministic AO graph found by the AO* algorithm is an acyclic subgraph, that represents how a problem is decomposed into smaller problems. However, for MDPs which may have cyclic policies, a different approach needs to be taken. Hansen and Zilberstein consider AO graphs with *loops* and present the LAO* algorithm. Loops will be present in any indefinite horizon MDP, since cycling over them is what can delay the solution indefinitely.

The AO* algorithm, described in Section 6.3, maintains a subgraph at all times. This subgraph is initialized to the given (start) node and contains *markers* along the direction of the best solution found as judged by the heuristic function. A marker indicates the hyper-edge to be followed at that node. That is, for the SSP it specifies the action to be taken. The heuristic function $h(n)$ for any node is an estimate of reaching the goal (solved) state starting from that node. It corresponds to the estimate of the value function for that node. The algorithm has two phases. In the forward phase, one travels down the marked path and

expands one of the *unsolved leaf* nodes, and assigns the heuristic value to each new node added to the subgraph. In the backward phase, these new values are backed up to the root (start) node. The process continues till the leaf nodes are all solved nodes, and no unsolved nodes remain in the subgraph.

The LAO^* algorithm maintains a subgraph G' made of a subset $S' \subseteq S$ of nodes from the state space, and on termination has actions prescribed for each node $n \in G_P'$. A partial solution graph G_P' is a subgraph of G' that contains exactly one action for each state in G' . The algorithm maintains the best partial solution at all times, indicated by the action markers. A partial solution graph is a solution graph, if it has no nonterminal leaf nodes.

The prescription of actions in the partial policy π' : $S' \rightarrow A$ is akin to the markers that the AO^* algorithm maintains.

In the forward phase, LAO^* follows the marked path and expands a non-terminal leaf state. A nonterminal leaf state is a leaf state that is not a goal state. Note that the leaf state cannot be a state that is already in G_P' , which would be the case with loops. The set of nodes eligible for expansion is called the *fringe* of the graph G_P' . Expansion of the nonterminal leaf state is done by adding the hyper-edges corresponding to each action, and evaluating the heuristic values of the new nodes. If a successor is already in the graph G_P' that is being constructed, that is there exists a loop, then it is not added again.

The main difference between AO^* and LAO^* is that instead of using a simple backup rule that AO^* uses for cost revision, the LAO^* uses a dynamic programming procedure like *Value Iteration* or *Policy Iteration*. This is required because the expected cost of executing an action depends upon the probabilities of each edge in the hyper-edge being executed, and the presence of loops.

The algorithm LAO^* adapted from (Hansen and Zilberstein, 2001) is described at a high level in Figure 17.38. In the algorithm, G is the graph implicitly rooted at the start state, and G_P represents the best partial policy found so far. This is equivalent to saying that G_P is the subgraph rooted at the start node and marked as the best choice at each node. In the forward phase, the algorithm picks a node from the fringe of the best partial solution and expands it (like in AO^*). In the backward phase (lines 9–10), it adopts a dynamic programming approach to update costs. Taking a cue from (Mausam and Kolobov, 2012), we have restricted this dynamic programming to *Policy Iteration*. The interested reader is referred to (Hansen and Zilberstein, 2001) for a version that includes *Value Iteration* as well.

```

LAO* (state space  $S$ , start state  $Start$ , action set  $A$ , costs  $C$ ,  $\epsilon$ )
1   $G \leftarrow Start$ 
2   $G_p \leftarrow G$ 
3   $Fringe \leftarrow \{Start\}$  /* Forward phase */
4  while  $Fringe \neq \{\}$ 
5      do remove some node  $N$  from  $Fringe$ 
6          expand  $N$  and add its new children  $C$  to  $G$ 
7          compute the heuristic value of each child in  $C$ 
8      /* Cost revision */
9       $Z \leftarrow \{N\} \cup \{\text{ancestors of } N \text{ in } G_p\}$ 
10     perform Policy Iteration on  $Z$ 
11 /* back to Forward phase */
12      $G_p \leftarrow$  subgraph with hyper-edges marked by best partial policy
13      $Fringe \leftarrow$  leaves of  $G_p$  that are not goal states
14 return  $G_p$ 

```

Figure 17.38 Like AO^* , the algorithm LAO^* works in two phases. In the forward phase, it follows hyper-edges marked by the best partial policy, expands a nonterminal node, and evaluates the heuristic values of the children. In the backward phase, it adopts a different approach for cost revision, since the graph may have loops. The cost revision process used here is Policy Iteration. The algorithm terminates when the graph for the best policy has no leaf nodes.

Observe that the set Z on which dynamic programming is done to solve for values includes (line 9) only the expanded node N and its ancestors in the solution graph being constructed. It excludes the children C of node N . The reason for not including the children is that *their* value is determined by the heuristic function and cannot be changed in the process of running the dynamic programming procedure⁴¹. At the same time, the children *do* affect the value of N during the procedure, and through N , the values get propagated to the ancestors of N as well.

The LAO^* algorithm takes a heuristic search approach to solving SSPs in which a start node has been specified. It employs a heuristic function to explore only that part of the state space which the heuristic function estimates to be cheapest. At the termination of the algorithm it finds the optimal policy π^* . This policy may be cyclic in nature.

We look at the initial steps LAO^* would take with our problem described in Figure 17.29 and Table 17.13. The transition graph it explores has been depicted in Figure 17.37. It begins with the start node s_1 . Since this is the only leaf node, it expands it with the possible actions a_{12} , a_{13} , a_{14} and a_{15} . This results in addition of the nodes s_2 , s_3 , s_4 and s_5 to the transition graph. As shown in Figure 17.39, the heuristic values of these four nodes are computed. The heuristic values we have used are the costs for deterministic paths from each node to the goal s_G , as determined from Figure 17.29. The backed up cost for the node s_1 is $V^{a_{15}}(s_1) = 21.7$, which is backed up from the hyper-edge for action a_{15} . This is the minimum of the possible values from each of the four actions as is shown in the figure. The graph G_p is shown with dark solid lines, and contains the marked path along the hyper-edge for the actions a_{15} .

The *fringe* is the leaves in G_p , that is, the nodes s_4 and s_5 . LAO^* will pick one of them for expansion. While in general, any node from the *fringe* can be chosen for expansion, in practice it helps to make informed choices. Two approaches would be to either choose a node that has the

highest probability of being reached, or the one that seems to be the cheapest.

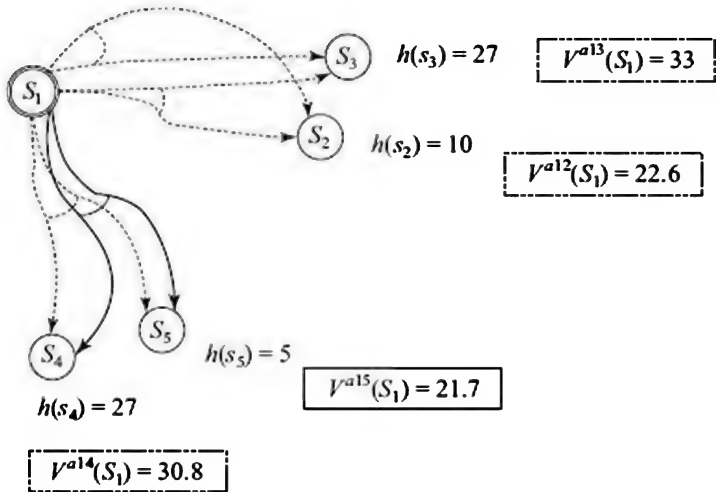


Figure 17.39 In the first cycle LAO* expands S_1 , the only node on the fringe. There are four actions a_{12} , a_{13} , a_{14} and a_{15} possible. Based on the costs in Figure 17.29 and the probabilities in Table 17.13, the best value that node S_1 gets is by action a_{15} , marked by the solid arrows. The fringe now is S_4 and S_5 , and LAO* will expand one of them.

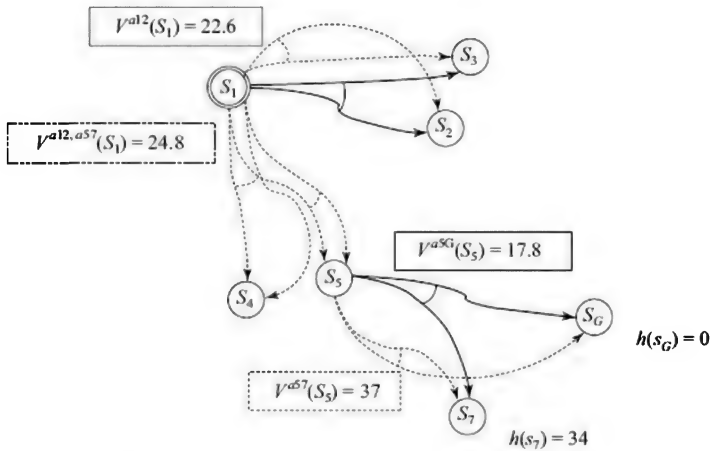


Figure 17.40 Expanding the fringe node S_5 from Figure 17.39, there are two possible actions a_{56} and a_{57} . The better one increases the expected cost (value) of S_5 to 17.8 and the corresponding value for S_1 to 24.2. LAO* marks the better options as shown by solid arrows. The forward phase now leads it to the fringe nodes S_2 and S_3 , one of which it will expand next.

We assume that the node s_5 is picked for expansion next. The resulting graph is depicted in Figure 17.40. The two hyper-edges emanating from s_5 , correspond to the two actions a_{56} and a_{57} , both leading to nodes s_6 and s_7 , but with different probabilities. The heuristic

values are $h(s_G) = 0$ and $h(s_7) = 34$ obtained, again, by finding the deterministic least cost. The estimate of s_7 may seem high, but one must remember that it is a long way from the goal node. The actual value is likely to be higher still. The best value for node s_5 comes, not surprisingly, from action a_{5G} and results in a value of 17.8, which is greater than the heuristic estimate of 15. This has to be propagated back into G_P and results in a revised value $V^{a_{15}, a_{57}}(s_1) = 24.8$ for the start node.

This revised value is not the lowest now. The lowest value is $V^{a_{12}}(s_1) = 22.6$ corresponding to the action a_{12} . The new Fringe is shown by the dark solid nodes, s_2 and s_3 . LAO* will next pick one of them.

We will leave the action at this stage. The process of adding new nodes will continue. After (say) s_2 is expanded one more, the node s_6 will join the fray too. Then depending upon which partial policy looks better, AO* will continue expanding the fringe nodes till it has the solution graph. The reader is encouraged to continue this hand simulation of the algorithm,

LAO* will, like AO*, find the optimal policy provided that the heuristic function is admissible. That is, $h(s) \leq V^*(s)$, the optimal value of the node. The values used in our example, which are the costs of the shortest deterministic paths, are indeed admissible, because the values of the states cannot go below these costs.

17.7 Combining Evidences to form Beliefs

Where do our beliefs come from?

According to the *Nyaya Sutras* of Gotama, there are four possible sources of knowledge—perception (*pratyaksha*), inference (*anumāna*), comparison (*upamāna*) and verbal testimony (*shabda*, which could be of God from the *vedās*, or of a trustworthy human!) (Vidyabhusana, 2003).

Quine and Ullian (1978) concur. They assert that the two main sources of our beliefs are direct observation and inference. They ask the question as to how we come to believe that there are quarks, chromosomes and nebula when none of them are directly perceptible. Obviously, we form theories and models of the world we live in and at times we have to revise our models (beliefs) when new evidence comes to light. Starting with the belief in the earth being flat and at the centre of the universe, we now accept that it is one on the nine (or eight? or ten?) planets going around our sun, which itself is a not too significant star in our galaxy.

In their book *The Web of Belief*, they describe a problem of updating beliefs when more evidence comes to the fore, with the following murder mystery (Quine and Ullian, 1978). “*Let Abbot, Babbitt, and Cabot be suspects in a murder case. Abbott has an alibi, in the register of a respectable hotel in Albany. Babbitt also has an alibi, for his brother-in-law testified that Babbitt was visiting him in Brooklyn at that time. Cabot pleads alibi too, claiming to have been watching a ski meet in the*

Catskills, but we have only his word for that."

It is clear that posing the above problem in logic, one would easily arrive at a conclusion. For example one could pose it as,

$$\begin{aligned} & \text{Murderer}(\text{abbott}) \vee \text{Murderer}(\text{babbitt}) \vee \text{Murderer}(\text{cabot}) \\ & \forall x (\text{ValidAlibi}(x) \supset \neg \text{Murderer}(x)) \\ & \forall x \forall y (\text{Murderer}(x) \wedge x \neq y \supset \neg \text{Murderer}(y)) \\ & \text{ValidAlibi}(\text{abbott}) \\ & \text{ValidAlibi}(\text{babbitt}) \end{aligned}$$

The reader can verify that the above sentences lead us to believe unequivocally that Cabot is the murderer.

However, there is a twist in the tale. "*But presently Cabot documents his alibi—he had the good luck to have been caught by television in the sidelines at the ski meet.*"

The reader should verify that now this results in a contradiction. The set of statements is inconsistent. And logic cannot deal with inconsistency (see Section 12.3.4). Holding all the given statements to be unequivocally true is not tenable. We know that one of the alibis must be false or there must be a hitherto unmentioned person. But given the information that we have, we must be deal with the statements with a flexible amount of belief in them. How valid would an alibi provided by a brother-in-law would be? Could Abbott have an accomplice working in the hotel? We know each of them are either *true* or *false*, but do not know enough to decide the truth values.

One approach to dealing with such problems is to use the probabilistic approach. In doing so, one would assign some prior probabilities to the three suspects (being the murderer). For example we could say,

$$P(A\text{-murderer}) = P(B\text{-murderer}) = P(C\text{-murderer}) = 1/3$$

Then we could assign degrees of belief to the possibility that they have a valid alibi, $P(A\text{-alibi})$, $P(B\text{-alibi})$ and $P(C\text{-alibi})$. Then we could relate the alibis to the probability of them being murderers, computing the posterior probabilities of the form $P(X\text{-murderer} \mid X\text{-alibi})$. Then we could make explicit the relation between a valid alibi to the reliability of the source.

In general, such a process of investigation would require collecting together evidence from diverse sources and combining them into a final conclusion. In the Bayesian framework, one would have to set up the different conditional probabilities and populate them with values. The approach would find it difficult to come to conclusions with partial incomplete evidences.

One approach that allows us to combine evidences together as and when they come is the *Dempster-Shafer theory of evidential reasoning*, which is a generalization of Bayesian theory of subjective probability (Dempster, 1968), (Shafer, 1976).

17.7.1 The Dempster-Shafer Theory

The Dempster-Shafer (D-S) theory of evidential reasoning is essentially a theory of combining evidences from different sources into one coherent belief system. Unlike the probabilistic approach, it does not require the complete set of prior and conditional probabilities. Let us say that we are investigating a set of exhaustive and mutually exclusive set hypotheses. The set of these hypotheses is called the *frame of discernment*. Assuming that there are k hypotheses, the frame of discernment is $\Theta = \{h_1, h_2, \dots, h_k\}$. The set is exhaustive in that no hypothesis has been left out, and it is mutually exclusive, in that only one hypothesis is true. In diagnosis terminology, we are assuming that there is a single fault. In medical terms, the patient is suffering from exactly one of the diseases in the frame of discernment. From a detective's point of view, there is only one culprit, and she is present in the set. We illustrate the theory with the following (fictitious) story:

In a certain country, the government had decided that cartoons lampooning persons in authority are not to be allowed. Yet in a certain college one day, a cartoon was found on the notice board. The cartoon made fun at the slow pace at which a senior teacher was grading examination papers. The senior teacher found it quite funny and laughed it off, explaining that the grading had to be done thoroughly. But some of the class teachers would have none of it. They assumed it must be one of the four friends in the art class, Aditi, Amala, Kopal, and Urvi, stated here in alphabetic order and not based on the class teachers' degree of suspicion, and started the investigation. M (we will not name the teachers) said that she was 60% sure that it was Aditi or Urvi because they were good at drawing. K said that he was 80% sure that it was Urvi or Amala because they were known to be girls of free spirit. P said that there was a 50% chance that it was Kopal or Amala, because he had seen them laughing near the water cooler.

Before we discuss the D-S theory, let us recap how the Bayesian approach would proceed. First, we would need the prior probabilities of the four students having drawn the cartoon. Then, we would need the likelihoods that if each had drawn the cartoon, they would have put it up. Then we would compute the posterior probabilities of each having drawn the cartoon, knowing that it had been put up. It is not clear how the evidence (opinion?) of the class teachers would be combined. Further there are hidden pitfalls in propagating changing posterior values that are beyond the scope of this text to explore. The interested reader is referred to (Pearl, 1988) for an indepth investigation.

The D-S theory operates with the power set of the frame of discernment and begins by assigning the entire belief or *mass* function to Θ . Then as more evidence arrives, it distributes the *mass* to the other

sets.

Let us take each of the three statements as pieces of evidence. Taken individually, they result in the following mass functions.

$$\begin{array}{lll} m_1(\{\text{Aditi, Urvi}\}) = 0.6 & \text{and} & m_1(\Theta) = 0.4 \\ m_2(\{\text{Amala, Urvi}\}) = 0.8 & \text{and} & m_2(\Theta) = 0.2 \\ m_3(\{\text{Amala, Kopal}\}) = 0.5 & \text{and} & m_3(\Theta) = 0.5 \end{array}$$

Observe that the three mass functions allocate the remaining mass to Θ . Thus, the first statement assigns a belief mass of 0.6 to the set $\{\text{Aditi, Urvi}\}$ and 0.4 to Θ . This amounts to saying that there is 0.6 belief in the statement $S = \text{"One of Aditi or Urvi is the culprit"}$, but it does not assign the remaining belief to $\neg S$ (which would be equivalent to saying that "One of Amala or Kopal is the culprit"). Instead, it assigns the remaining mass to Θ which amounts to saying that "One of Aditi, Amala, Kopal or Urvi is the culprit", which is equivalent to saying that we do not know who the culprit is.

This is one way the D-S theory differs from probability theory. If in probability theory one would have said that $P(S) = 0.6$ then it would necessarily mean that $P(\neg S) = 0.4$ because the two must sum to 1. The D-S theory assigns the belief mass to the statement pertaining to the evidence, and assigns the remaining to the frame of discernment.

The power set of Θ is the set of possible statements one can assign belief to, each set being a disjunction of culprits (or a conjunction of suspects). The belief assigned to each set $\text{Bel}(A)$, $A \subseteq \Theta$, is the sum of belief mass of all its subsets.

$$\text{Bel}(A) = \sum_{B \subseteq A} m(B)$$

When the first piece of evidence comes in, the mass distribution is $m_1(\{\text{Aditi, Urvi}\}) = 0.6$ and $m_1(\Theta) = 0.4$. This corresponds to $\text{Bel}(\{\text{Aditi, Urvi}\}) = 0.6$, $\text{Bel}(\Theta) = 1.0$, and for every set S such that $\{\text{Aditi, Urvi}\} \subseteq S$, $\text{Bel}(S) = 0.6$. The rest have zero belief.

$$m_{jk}(A) = m_j \oplus m_k(A) = \sum_{X \cap Y = A} m_j(X) m_k(Y)$$

When there are more than one pieces of evidence, one can use the *Dempster's rule* to combine the evidences. Let m_j and m_k be two mass functions. Then the combined mass function $m_k = m_j \oplus m_k$, given by Dempster's rule, is as follows.

That is, after combining the two sets of evidences with belief mass m_j and m_k , the updated mass for any set A is the sum of the products of $m_j(X)$ and $m_k(Y)$ for all pairs X and Y such that $X \cap Y = A$. In the simple case (as in our example) when there is only one such pair, the summation is not needed, and is

The four sets which have the evidence to be combined are $\{\text{Aditi, Urvi}\}$ and Θ for m_2 , and $\{\text{Amala, Urvi}\}$ and Θ for m_2 . The table below contains all no empty intersections for which the Dempster's rule results

in nonzero mass.

Table 17.15 Combining the first two set evidences

$m_{12} = m_1 \oplus m_2$	$m_1(\{\text{Aditi, Urvi}\}) = 0.6$	$m_1(\Theta) = 0.4$
$m_2(\{\text{Amala, Urvi}\}) = 0.8$	$m_{12}(\{\text{Urvi}\}) = 0.48$	$m_{12}(\{\text{Amala, Urvi}\}) = 0.32$
$M_2(\Theta) = 0.2$	$m_{12}(\{\text{Aditi, Urvi}\}) = 0.12$	$m_{12}(\Theta) = 0.08$

Thus we can see that the belief mass has been redistributed to four sets—{Urvi}, {Aditi, Urvi}, {Amala, Urvi} and Θ . The updated distribution is shown in Figure 17.41.

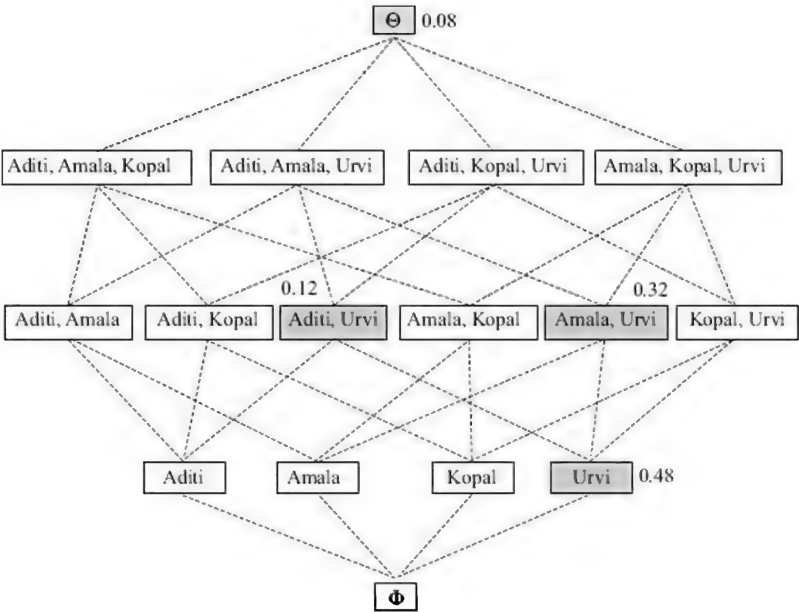


Figure 17.41 The belief mass distribution after two statements have been combined. For any set A , the belief $\text{Bel}(A)$ is the sum of beliefs of all its subsets. Observe that $\text{Bel}(\{\text{Aditi, Urvi}\})$ is still 0.6 and $\text{Bel}(\{\text{Amala, Urvi}\})$ is 0.8.

The Dempster-Shafer theory defines the amount of belief in any set A as an interval defined by two values. One, $\text{Bel}(A)$ is the minimum belief one has in the set. The other $P(A)$, or plausibility of A , defines the maximum possible belief in the set A . The resulting interval $\langle \text{Bel}(A), P(A) \rangle$ is defined as the confidence interval. For a consistent set of beliefs $\text{Bel}(A) < P(A)$. The plausibility of A , $P(A)$ is defined as,

$$P(A) = 1 - \text{Bel}(A^c)$$

where A^c is the complement of set A with respect to Θ . The confidence intervals for the four sets in our example are given below. Recall that the

belief in a set is the sum of belief mass of all its subsets.

$$\begin{aligned}\text{Bel}(\{\text{Urvi}\}) &= m_{12}(\{\text{Urvi}\}) = 0.48 \\ P(\{\text{Urvi}\}) &= 1 - \text{Bel}(\{\text{Aditi}, \text{Amala}, \text{Kopal}\}) = 1 - 0 = 1\end{aligned}$$

The confidence interval for $\{\text{Urvi}\}$ is $\langle 0.48, 1.0 \rangle$, saying that the minimum belief in the hypothesis is 0.48 and the maximum possible belief is 1.0. Likewise,

$$\begin{aligned}\text{Bel}(\{\text{Aditi}, \text{Urvi}\}) &= m_{12}(\{\text{Aditi}, \text{Urvi}\}) + m_{12}(\{\text{Urvi}\}) = 0.12 + 0.48 = 0.6 \\ P(\{\text{Aditi}, \text{Urvi}\}) &= 1 - \text{Bel}(\{\text{Amala}, \text{Kopal}\}) = 1 - 0 = 1 \\ \text{Bel}(\{\text{Amala}, \text{Urvi}\}) &= m_{12}(\{\text{Amala}, \text{Urvi}\}) + m_{12}(\{\text{Urvi}\}) = 0.32 + 0.48 = 0.8 \\ P(\{\text{Amala}, \text{Urvi}\}) &= 1 - \text{Bel}(\{\text{Aditi}, \text{Kopal}\}) = 1 - 0 = 1 \\ \text{Bel}(\Theta) &= m_{12}(\{\text{Aditi}, \text{Urvi}\}) + m_{12}(\{\text{Urvi}\}) + m_{12}(\{\text{Amala}, \text{Urvi}\}) + m_{12}(\Theta) \\ &= 0.12 + 0.48 + 0.32 + 0.08 = 1.0 \\ P(\Theta) &= 1 - \text{Bel}(\Phi) = 1 - 0 = 1\end{aligned}$$

Observe that the overall beliefs in the two sets for which evidence was received $\{\text{Aditi}, \text{Urvi}\}$ and $\{\text{Amala}, \text{Urvi}\}$ remained unchanged after combining the two evidences.

Let us compare this to reasoning with logic, with the assumption that the statements are hundred percent true. The problem could be stated as,

$$\begin{aligned}\text{Cartoonist}(\text{Aditi}) \vee \text{Cartoonist}(\text{Amala}) \vee \text{Cartoonist}(\text{Kopal}) \vee \text{Cartoonist}(\text{Urvi}) \\ \text{Cartoonist}(\text{Aditi}) \vee \text{Cartoonist}(\text{Urvi}) \\ \text{Cartoonist}(\text{Amala}) \vee \text{Cartoonist}(\text{Urvi}) \\ \forall x \forall y (\text{Cartoonist}(x) \wedge x \neq y \supset \neg \text{Cartoonist}(y))\end{aligned}$$

Then the only conclusion that one can draw is that Urvi is the one who put up the cartoon. Given that the statements are not a hundred percent true and one has a partial belief in each, the D-S theory results in a partial belief in the same statement. Observe that $\{\text{Urvi}\}$ is the only singleton set (which satisfies the mutual exclusion property) that has positive belief.

Now consider the third piece of evidence that comes in the points to the set $\{\text{Amala}, \text{Kopal}\}$, with belief mass 0.5. In the logic framework if one were to treat it as a true statement and add $(\text{Cartoonist}(\text{Amala}) \vee \text{Cartoonist}(\text{Kopal}))$ to the set of statements, it would result in a contradiction. The D-S theory updates the belief mass distribution as follows.

Table 17.16 The Dempster combination rule after the third piece of evidence

$m_{123} = m_{12} \oplus m_3$	$m_3(\{\text{Amala}, \text{Kopal}\}) = 0.5$	$m_3(\Theta) = 0.08$
$m_{12}(\{\text{Urvi}\}) = 0.48$	$m_{123}(\Phi) = 0.24$	$m_{123}(\{\text{Urvi}\}) = 0.24$
$m_{12}(\{\text{Aditi}, \text{Urvi}\}) = 0.12$	$m_{123}(\Phi) = 0.06$	$m_{123}(\{\text{Aditi}, \text{Urvi}\}) = 0.06$
$m_{12}(\{\text{Amala}, \text{Urvi}\}) =$		$m_{123}(\{\text{Amala}, \text{Urvi}\}) =$

0.32	$M_{123}(\{\text{Amala}\}) = 0.16$	0.16
$m_{12}(\Theta) = 0.08$	$m_{123}(\{\text{Amala}, \text{Kopal}\}) = 0.04$	$m_{123}(\Theta) = 0.04$

The first thing to observe is that the empty set signifying inconsistency has acquired a nonzero mass ($0.24 + 0.06 = 0.30$). This inconsistency, as we have seen above, is due to the (somewhat unprincipled and) inconsistent allegations made against different sets of students. The resulting mass distribution is depicted in Figure 17.42.

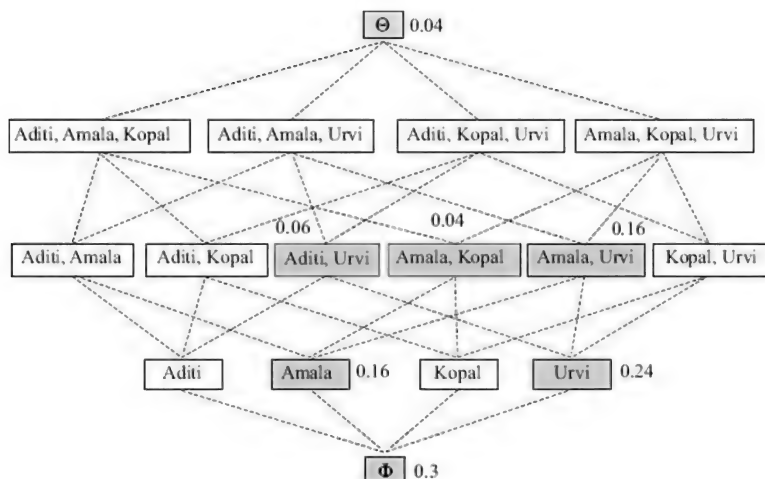


Figure 17.42 The final belief mass distribution after the three statements have been combined. Observe that with the third piece of evidence, some mass has gone to the empty set which signify a degree of inconsistency.

Let us compute the confidence intervals for the resulting sets,

$$\begin{aligned}
 \text{Bel}(\{\text{Urvi}\}) &= m_{123}(\{\text{Urvi}\}) + m_{123}(\Phi) = 0.24 + 0.3 = 0.54 \\
 P(\{\text{Urvi}\}) &= 1 - \text{Bel}(\{\text{Aditi}, \text{Amala}, \text{Kopal}\}) \\
 &= 1 - (m_{123}(\{\text{Amala}\}) + m_{123}(\{\text{Amala}, \text{Kopal}\}) + m_{123}(\Phi)) \\
 &= 1 - (0.16 + 0.04 + 0.30) = 1 - 0.50 = 0.50
 \end{aligned}$$

The confidence interval for $\{\text{Urvi}\}$ has now become $\langle 0.54, 0.5 \rangle$, which signals inconsistency. The remaining confidence intervals are,

$$\begin{aligned}
\text{Bel}(\{\text{Amala}\}) &= m_{123}(\{\text{Amala}\}) + m_{123}(\Phi) = 0.16 + 0.3 = 0.46 \\
P(\{\text{Amala}\}) &= 1 - \text{Bel}(\{\text{Aditi}, \text{Kopal}, \text{Urvi}\}) \\
&= 1 - (m_{123}(\{\text{Urvi}\}) + m_{123}(\{\text{Aditi}, \text{Urvi}\}) + m_{123}(\Phi)) \\
&= 1 - (0.24 + 0.06 + 0.30) = 1 - 0.60 = 0.40 \\
\text{Bel}(\{\text{Aditi}, \text{Urvi}\}) &= m_{123}(\{\text{Aditi}, \text{Urvi}\}) + m_{123}(\{\text{Urvi}\}) + m_{123}(\Phi) \\
&= 0.06 + 0.24 + 0.3 = 0.6 \\
P(\{\text{Aditi}, \text{Urvi}\}) &= 1 - \text{Bel}(\{\text{Amala}, \text{Kopal}\}) \\
&= 1 - (m_{123}(\{\text{Amala}\}) + m_{123}(\{\text{Amala}, \text{Kopal}\}) + m_{123}(\Phi)) \\
&= 1 - (0.16 + 0.04 + 0.30) = 1 - 0.50 = 0.50 \\
\text{Bel}(\{\text{Amala}, \text{Urvi}\}) &= m_{123}(\{\text{Amala}, \text{Urvi}\}) + m_{123}(\{\text{Amala}\}) + m_{123}(\{\text{Urvi}\}) + m_{123}(\Phi) \\
&= 0.16 + 0.16 + 0.24 + 0.3 = 0.86 \\
P(\{\text{Amala}, \text{Urvi}\}) &= 1 - \text{Bel}(\{\text{Aditi}, \text{Kopal}\}) \\
&= 1 - m_{123}(\Phi) = 1 - 0.3 = 0.7 \\
\text{Bel}(\{\text{Amala}, \text{Kopal}\}) &= m_{123}(\{\text{Amala}, \text{Kopal}\}) + m_{123}(\{\text{Amala}\}) + m_{123}(\Phi) \\
&= 0.04 + 0.16 + 0.3 = 0.50 \\
P(\{\text{Amala}, \text{Kopal}\}) &= 1 - \text{Bel}(\{\text{Aditi}, \text{Kopal}\}) \\
&= 1 - m_{123}(\Phi) = 1 - 0.3 = 0.7 \\
\text{Bel}(\Theta) &= 1.0 \\
P(\Theta) &= 1 - \text{Bel}(\Phi) = 1 - 0.3 = 0.7
\end{aligned}$$

One can observe that in every case above the plausibility (which signifies maximum possible belief) has gone below the belief. It has been suggested that an alternate formula for computing be used in which one could normalize away the inconsistency. In the revised formula, one does not count the belief assigned to the empty hypothesis, but instead divides the beliefs obtained by a normalizing factor that is proportional to the degree of inconsistency. This could result in some counter-intuitive results that mask the inconsistency though (see Exercise 22). The revised Dempster's rule is as follows,

$$m_{jk}(A) = m_j \oplus m_k(A) = \frac{1}{1-K} \sum_{X \cap Y = A \text{ and } A \neq \Phi} m_j(X) m_k(Y)$$

where K is the total belief in Φ ,

$$K = \sum_{X \cap Y = \Phi} m_j(X) m_k(Y)$$

The reader is encouraged to apply the revised formula to the example above.

17.8 Discussion

In this chapter we have discussed some techniques that have been proposed to deal with uncertainty. In the real world, any agent is faced with uncertainty and has to deal with in some way. The various approaches discussed here involve default reasoning, assumption based reasoning, qualitative reasoning, abduction and probabilistic approaches to characterize uncertainty. Every approach requires its own

representation and reasoning schemes.

Human beings deal with uncertainty in a variety of ways. Apart from first principles mechanisms to reason with uncertain knowledge described in this chapter, we also exploit experience and knowledge gleaned from other sources. We do not always *solve* the current problem, but often rely on memory to produce a solution. Some of these techniques have been discussed in earlier chapters. Even such approaches require issues in representation and reasoning (principally retrieval) to be addressed.

The ontological and epistemological problems associated with knowledge representation are the key to artificial intelligence. How does an agent structure knowledge and how does it acquire the knowledge? In the next chapter, we look at one of the last frontiers of AI research—machine learning. In the end, the agent has to acquire knowledge on its own, from its experiences, from being taught, by observing and generalizing. The ability to augment its knowledge base would be a key to building an autonomous, intelligent agent.



Exercises

1. Given the knowledge base below,

```
KB: {  ∀x ∀y (Friend(x,y) ⊃ Friend(y,x))
      ∀x ∀y (Friend(x,y) ∧ ¬Isolated(y) ⊃ Chat(x,y))
      Friend(aditi, shubhagata), Friend(aditi, jennifer),
      (Isolated(jennifer) ∨ Isolated(shubhagata))
}
```

what is the answer to the query $\exists x(\text{Chat}(\text{aditi}, x))$ using (a) the closed world assumption, and (b) $\text{Circ}[KB; \text{Isolated}]$? What about $\text{Chat}(\text{aditi}, \text{shubhagata}) \vee \text{Chat}(\text{aditi}, \text{jennifer})$?

2. Given the following default theory,

```
Fi = { Friend(aditi, shubhagata), Friend(aditi, jennifer),
      (Isolated(jennifer) ∨ Isolated(shubha)) }
Di = { <Friend(aditi, shubha): ¬Isolated(aditi) ∧ ¬Isolated(shubha) /
      Chat(aditi, shubha)>
      <Friend(aditi, jennifer): ¬Isolated(aditi) ∧ ¬Isolated(jennifer) /
      Chat(aditi, jennifer)>
}
```

Construct all extensions of the default theory. What conclusions about the *Chat* predicate can be drawn?

3. Generate all the expansions of the following knowledge base and identify the stable ones

```
KB: { ∀x ∀y (Friend(x,y) ∧ ¬BIssolated(x) ∧ ¬BIssolated(y) ⊃ Chat(x,y))
      Friend(aditi, jennifer), Isolated(jennifer)
}
```

4. Discuss the implications of replacing the universal statement with the following one

$$\mid \forall x \forall y (\text{Friend}(x,y) \wedge \mathbf{B}\neg\text{Isolated}(x) \wedge \mathbf{B}\neg\text{Isolated}(y) \supset \text{Chat}(x,y))$$

in the knowledge base KB_5 reproduced below. Construct all stable expansions and identify the stable ones.

```
KB5: {   $\forall x \forall y ( \text{Friend}(x,y) \wedge \neg\mathbf{B}\text{Isolated}(x) \wedge \neg\mathbf{B}\text{Isolated}(y) \supset \text{Chat}(x,y) )$ 
      Friend(aditi, shubhgata), Friend(aditi, jennifer),
      Isolated(jennifer)
}
```

5. Consider the following facts that Raymond might be working with,

```
{  $\forall x ( \text{Lying}(x) \supset \text{GetQuestioned}(x) ,$ 
   $(\neg\mathbf{B}\text{Lying}(\text{shashi}) \supset \text{Lying}(\text{lalit})) , (\neg\mathbf{B}\text{Lying}(\text{lalit}) \supset \text{Lying}(\text{shashi}))$ 
}
```

Who do you think should “get questioned”? Do you think this KB allows for the fact that both might be lying, or neither?

6. What are *stable sets* in the context of autoepistemic logic? How are they constructed? What are the stable sets for the following sets of sentences?

(a) $\{(\neg\mathbf{B}\text{Flat}(\text{Earth}) \supset \text{Flat}(\text{Earth})), \text{Hot}(\text{Sun}), \text{Round}(\text{Moon})\}$

(b) $\{(\mathbf{B}\text{Flat}(\text{Earth}) \supset \text{Flat}(\text{Earth})), \text{Hot}(\text{Sun}), \text{Round}(\text{Moon})\}$

7. Given the following set of statements,

```
Bird(tweety), Bird(chilly), Bird(chirpy), Myna(chirpy)
chirpy $\neq$ chilly, chirpy $\neq$ tweety, chilly  $\neq$ tweety,
Penguin(chirpy)  $\vee$  Penguin(chilly)
 $\forall x ( \text{Penguin}(x) \supset \neg\text{Flies}(x) )$ 
 $\forall x ( \text{Penguin}(x) \equiv \neg\text{Myna}(x) )$ 
```

Express the statement “In general, birds fly” and show how the queries “Flies(tweety)?”, “Flies(chilly)?”, “Flies(chirpy)?” are answered using,

(a) Circumscription

(b) Autoepistemic logic

8. In the following problem (due to Kenneth Forbus), a ship generates superheated steam to drive its propulsion system. Water at sea temperature T_{Input} is taken into a boiler and steam is generated.

This steam is pushed through a superheater that increases the temperature to around 500° Celsius. Let us say that the ship travels from the Arctic seas to the tropics, so that the water is taken in at a higher temperature. Does this affect the output temperature T_{Output} ? If yes, how?

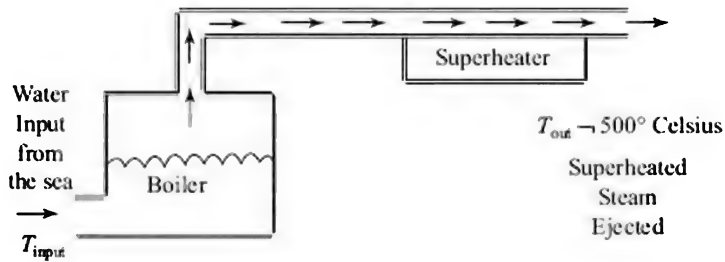


Figure 17.43 The propulsion system of a steam ship.

9. In the scenario depicted in Section 17.2, assume that the walls in the room have windows that are open. How will the qualitative analysis of the moving ball change as a consequence?
10. Create an envisionment similar to the Figure 17.5, for a ball thrown at an angle in a two-dimensional room without windows.
11. In the above envisionment, represent the different qualitative states using the $\langle qmag, qdir \rangle$ representation of QSIM.
12. Given the steady state of the two tank system described in Table 17.7, determine the qualitative behaviour of the system, if tank A were to be suddenly filled to the brim.
13. Given a lattice representing the power set of components, and a set of candidates for diagnosis in the lattice (as in Section 17.3.3), write an algorithm to accept a new conflict set and update the set of candidates for diagnosis.
14. Given the conflict $\langle M_1, M_2, A_1 \rangle$ and the observations $F_1=10$ and $F_2=12$ in Section 17.3.1, derive another constraint by deriving the value of C_2 in two different ways.
15. In the diagnosis example considered in the text (Figure 17.12), the discrepancy between the observed value $F_1=10$ and expected value $F_1=12$ led to the candidates $\{M_2, A_2\}$ and $\{M_3, A_2\}$. In what way could these diagnoses explain the observed discrepancy?
16. Write an algorithm that takes a set of candidate diagnoses and a new conflict and generates the new set of candidates.
17. After processing the input $F_1=10$ and $F_2=12$, one more measurement is made. Extend the diagram in Figure 17.15 to incorporate the results of the new measurement, in each of the following cases independently.
 - (a) $C_1 = 4$
 - (b) $C_1 = 6$
 - (c) $C_1 = 8$
 - (d) $C_2 = 4$
 - (e) $C_2 = 6$
 - (f) $C_2 = 8$
 - (g) $C_3 = 4$
18. Replace the contradiction node \perp in Figure 17.21 with a node containing the datum S, and reassign the labels to all the nodes.

19. Given that you have buckets red and green which you choose with probabilities 0.3 and 0.7. Let the red bucket have 90 black balls and 10 white balls, while the blue one has 10 black ones and 90 white ones. Suppose you choose a bucket randomly, as per the given probabilities, and randomly pick a ball from the bucket. What is the probability of picking up a black ball? Alternatively, if you *have* a black ball, which bucket did it most likely come from?
20. The birthday clash problem. How many people do you need in a room so that the probability of a birthday clash is greater than 0.5?
21. A variation on the three prisoners paradox from (Gardner, 1959a; 1959b) and (Pearl, 1988). Three prisoners⁴² A, B and C are in a jail to be tried for a massive XYZ scam. The judgement is to be out the next morning. Only one of them will be declared guilty, and the other two will be released. The prison guard knows the identity of the convicted person.

Prisoner A requests the guard to hand over a letter to one of the other two who is going to be released. The guard agrees and does so. Prisoner A then asks the guard as to whom he gave the letter to, and the guard informs him that it was to prisoner B.

Prisoner A reasons as follows, "Before I spoke to the guard, the chances of me being convicted were $1/3$. Now that I know that B is going to be released, only C and I are left. So the chances are now $1/2$. What did I do wrong?"

Is the prisoner A's reasoning correct?

22. The Monty Hall problem, a variation of the three prisoners paradox, is based on a game show hosted by Monty Hall (see http://en.wikipedia.org/wiki/Monty_Hall_problem).
You are shown three closed doors by the host of a game show, Monty. Behind one of the doors is a car (which you want) and behind the other two is a goat (which you don't want). Let us say you choose a door X. Then Monty opens another door Y ($\neq X$, he can always do that). You get a chance to switch to door Z ($\neq X, \neq Y$). The question is, do you gain from switching to Z?
23. Pose the problem in Section 17.6.1 to find the maximum *a posteriori* hypothesis problem as solve it. Assume the prior probabilities as $P(\text{Aditi}) = P(\text{Urvi}) = 0.3$ and $P(\text{Amala}) = P(\text{Kopal}) = 0.2$. Let the likelihoods of the cartoon being put up be $P(\text{Cartoon}|\text{Amala}) = P(\text{Cartoon}|\text{Urvi}) = 0.6$, $P(\text{Cartoon}|\text{Aditi}) = 0.5$ and $P(\text{Cartoon}|\text{Kopal}) = 0.55$. What is the most likely hypothesis? How will you incorporate the third piece of evidence?
24. Given that a reliable doctor believes that the patient has meningitis with probability 0.99 or a brain tumour with probability 0.01. Let another equally reputable doctor believe that the patient has suffered a concussion with probability 0.99 and brain tumour with probability 0.01. What is the most likely diagnosis obtained by combining the two pieces of evidence in the D-S approach with the revised (normalized) Dempster's rule for combining evidence?

—This problem was apparently posed by Lotfi Zadeh (1984) to show that the revised rule leads to a counterintuitive conclusion.

Also see http://en.wikipedia.org/wiki/Dempster%E2%80%93Shafer_theory

25. The adjoining Figure 17.44 depicts with shaded areas the time intervals your local news channel allocates to content. The rest of the time is devoted to advertisements. Assuming that we count time in minutes, the content durations are $\langle 0, 7 \rangle$, $\langle 10, 13 \rangle$, $\langle 17, 20 \rangle$, $\langle 21, 22 \rangle$ and similarly the next 30 minutes. The intervening gaps between the shaded areas are of 3, 4, 1 and 8 minutes every half an hour. Assuming that you switch on the television and are confronted with an advertisement, what is the likelihood that the time is in the interval of 8 minutes, starting at 22 minutes past the hour?

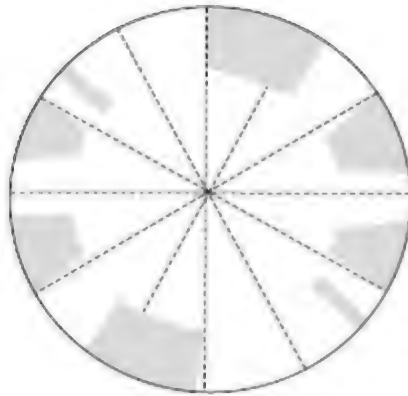


Figure 17.44 The shaded areas show the time interval when content is broadcast every hour.

26. Consider the transition graph in Figure 17.37. Consider it to be an indefinite horizon MDP with the following reward structure $R(s_G) = 100$, $R(s_6) = -100$ and all other states having zero reward. Let the costs of moves and probabilities be defined as the problem as 17.13. Pose the above problem as an SSP.
27. Consider the following stochastic shortest path problem described in (Hansen and Zilberstein, 2001). Figure 17.45 depicts a grid world in which the agent has to move from square 1 to square 4. The moves available are Up, Down, Left, and Right. Each move is applicable, if there exists a square in the direction of the move. Each move succeeds with a probability 0.5 and cost 1.
- Draw the transition graph for the above problem and show how LAO^* will solve it. [Caution: Treat self-loops carefully during cost revision.]
28. The Coverup Bureau said that either Mr. K or Mr. Y had stolen the goods. Its chief said he was 70% sure of it. The Cocktail Circuit said with 50% confidence that Mr. C had done no wrong. The Jumbo Circus claimed that it was either Mr. K or Mr. R with 80% certainty.

If you were to believe in what they said, who according to you is the prime culprit, and what is your degree of belief in your conclusion?

What is the confidence interval supporting the statement “It was either Mr. Y or Mr. R”? Justify all your answers.

29. Sherlock brooded over the piece of paper he had just received. “It is either Tinker, or Tailor”, he muttered finally, “but my sources are only 50% reliable”. “You could be right”, added Pradosh, “because my investigations reveal, with 0.8 confidence that it is either Tinker or Soldier or Spy”. They all looked at Hercule who, twirling his moustache, said, “I think that it is either Tailor or Spy. I am 60% certain of it.”

Everybody knew that one of Tinker, Tailor, Soldier and Spy had stolen the document, but none was quite sure who. After a while George, who had been listening intently said, “The evidence collected by you all is inconsistent. However, the best we can do is to conclude that ____ is the culprit.” The name was lost in thunder.

- (a) Who, according to *you*, is the culprit? Quantify your belief.
 (b) Is George’s remark of inconsistency justified?
 (c) What is your final belief in Sherlock’s statement?

2	3	4 Goal
1 Start		5
8	7	6

Figure 17.45 A grid world.

- (d) What is your belief in the statement that Soldier is not the culprit?
 (e) What is your belief in the statement that Spy is not the culprit?
 Give reasons for your answers.
30. Consider the Dempster-Shafer approach to evidential reasoning. Given a frame of discernment Θ , let there be three sets $A_1 \subset \Theta$, $A_2 \subset \Theta$, $A_3 \subset \Theta$ for which some evidence has been received with $m_1(A_1)$, $m_2(A_2)$ and $m_3(A_3)$ degrees of support. Given a set $A \subseteq \Theta$, how does one compute $Bel(A)$? What is the associated confidence interval?

¹ The sets containing the elements in the domain.

² Here the intent is to depict *isolation* as not having access to the internet, but isolation could come from other reasons as well. It depicts

- abnormality with respect to chatting with friends.
- 3 Observe that the quantification is over a predicate.
 - 4 Assuming that the set of predicates being circumscribed is identified.
 - 5 Or a special interest group on a social networking site.
 - 6 To be precise, we should also add all the tautologies of FOL to each extension because they are always entailed.
 - 7 In the literature, one can also find this expressed as Ka which stands for “ a is known”
 - 8 We treat the word *expansion* in the same manner as *extension* used earlier.
 - 9 Alternatively, we can say that in the stability criteria α does not contain the B operator.
 - 10 The reader might have heard the joke about a farmer seeking help from scientists and getting a letter in return that begins with “Consider a spherical cow... ”.
 - 11 And the *Sign* scale over $\{-, 0, +\}$ is coarsest of them all.
 - 12 <http://www.cs.utexas.edu/users/qr/QR-software.html>
 - 13 Observe that the value $(0, \infty)$ is equivalent to saying it is positive. If we had a landmark value, say the velocity of sound or the escape velocity on Earth, we could have used that as well.
 - 14 In the original U-tube diagram used by Kuipers (86; 94), the two tanks have a simple shape of uniform cross section. We use somewhat more exotic shapes only to emphasize the fact that qualitative reasoning does not need the quantitative data of volume.
 - 15 One could eschew the *Ab* predicate and simply infer $\neg \text{Multiplier}(M)$ with the understanding that M no longer qualifies to be called a multiplier, but the use of an explicit status predicate makes it neater. As we will see, that also makes it possible to introduce fault models easily. Other works have used $\neg \text{OK}(M)$ to mean the same thing as *Ab*(M).
 - 16 We will use the word *conflict* to refer to the conflict set as well.
 - 17 In the logic based model, each set should really stand for sets of statements asserting that the corresponding components are broken. We denote those statements by identifying the components for the sake of brevity.
 - 18 One could also think of exploring the value $F_2=12$ in our example as taking a second measurement as described in (Forbus and de Kleer, 1992).
 - 19 Or Ockham’s razor after William of Ockham (c. 1285–1349), Munich citizen, that “refers to distinguishing between two theories either by shaving away unnecessary assumptions or cutting apart two similar theories.” - Wikipedia, Occam’s razor.
 - 20 Interestingly, a model based diagnosis tool named RAZ’R is offered by OCC’M Software GmbH by two Munich citizens (see <http://www.occm.de/>).
 - 21 For example, a dam or a reservoir.
 - 22 The published system only looks at minimal sets of components. If one were to distinguish between *P-IN* failing by being blocked and by being perforated, then $\{P\text{-IN}, O\}$ would also appear as a candidate diagnosis.
 - 23 Doyle observes that the name Truth Maintenance is probably a

misnomer, but keeps it in order to be consistent with historical usage. Other names that have been suggested are Reason Maintenance and Consistency Maintenance.

24 This is similar to the notion of nogood used in solving constraint satisfaction problems, where a set of variables along with their tentative assignments may be marked as a nogood (see Chapter 9).

25 A well known Indian parliamentarian had said that “politics is the art of managing contradictions”!

26 (de Kleer, 1986) distinguishes between an assumption “A” and its datum “a” and represents the ATMS node as $\langle a, \{\{A\}, \{\{A\}\} \rangle$.

27 The following quote, and some variations of it, is attributed to the science fiction writer Arthur C Clarke (1917–2008)—“*Sometimes I think we’re alone in the universe, and sometimes I think we’re not. In either case, the idea is quite staggering.*”

28 <http://explorebiodiversity.com/Mexico/Pages/Habitats/species.htm>, accessed May 2012.

29 The reader may compare this with a definition of a constraint satisfaction problem defined in Chapter 9.

30 There is a temptation though to think of $P(\text{NewToy}, \text{Happy}) = P(\text{Happy} | \text{NewToy}) P(\text{NewToy})$ as being similar to Modus Ponens. But the reader must remember that this yields only a value of 0.4 because we cannot say that $P(\text{NewToy}) = 1$.

31 The author recalls with considerable relief the accurate diagnosis of a dear one by an experienced doctor, simply after hearing the major symptoms, even while a battery of tests were being conducted in a major hospital without success.

32 See <http://www.computing.surrey.ac.uk/ai/PROFILE/mycin.html#Expert>

33 In Pearl’s book, written in 1988, you hear this on the radio.

34 Notwithstanding the so called ‘Butterfly Effect’ in *Chaos Theory* that says that the flutter of a butterfly’s wings in China could affect the weather patterns in New York.

35 One could just as well have written $\mathbb{P}(X_1, \dots, X_7) = \mathbb{P}(X_7 | X_6 \dots X_1) * \mathbb{P}(X_6 | X_5 \dots X_1) * \dots * \mathbb{P}(X_2 | X_1) * \mathbb{P}(X_1)$ or in fact chosen the variables in any other order.

36 See Exercise 13 of Chapter 10 for a complex version of this game.

37 Named after the Russian mathematician, Andrey Andreyevich Markov (1856 - 1922).

38 We will use SSP as a short form for SSP MDP.

39 Legend has it that Robert Bruce, who was fighting for the freedom of Scotland, was inspired by watching a spider weaving her web who after failing for six times to throw a thread across a divide, persisted, and succeeded the seventh time.

40 See http://en.wikipedia.org/wiki/Bellman_equation

41 Andrey Kolobov, personal communication.

42 The names have been withheld and are anyway fictitious.

1976, Springer, pp. 306–320.

Black, 1990) Martha E. Pollack: *Plans as Complex Mental Attitudes*. In, P. R. Cohen, J. L. Morgan and E. Pollack (Eds.), *Intentions in Communication*, The MIT Press, Cambridge, MA.

Chen, 2010) Peter T. Poon, *Excellent Long-term Partnership between Voyager and the Deep Space Network*, NASA, JPL website, http://voyager.jpl.nasa.gov/news/profiles_dsn.html.

Porter, 1980) Porter, M. (1980): *An Algorithm for Suffix Stripping*. *Program*, 14(3), pp. 130–137.

Poundstone, 1993) William Poundstone, *Prisoner's Dilemma: John Von Neumann, Game Theory and the Puzzle of the Bomb*, Anchor; Reprint edition (January 1, 1993).

Price and Pegler, 1995) C.J. Price and I. Pegler: *Deciding Parameter Values with Case-based Reasoning*. In: I. Watson Editor, *Progress In Case-based Reasoning, Lecture Notes in Artificial Intelligence* 1020, Springer, Berlin.

Price et al., 1997) Chris J. Price, Ian S. Pegler, M. B. Ratcliffe, A. McManus: *From Troubleshooting to Process Design: Closing the Manufacturing Loop*. In David B. Leake, Enric Plaza (Eds.): *Case-based Reasoning Research and Development, Second International Conference, ICCBR-97*, Providence, Rhode Island, USA, July 25–27, 1997, *Proceedings. Lecture Notes in Computer Science* 114, Springer, pp. 114–121

Prud'hommeaux and Seaborne, 2008) Eric Prud'hommeaux and Andy Seaborne: *SPARQL Query Language for RDF, W3C Recommendation*, 15 January 2008. Available at <http://www.w3.org/TR/sparql-query/>.

Py et al., 2010) F. Py, K. Rajan and C. McGann: *A Systematic Agent Framework for Situated Autonomous Systems*. In *Proceedings, 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*.

Quillian, 1969) M. Ross Quillian: *The Teachable Language Comprehender: A Simulation Program in the Theory of Language*. *Commun. ACM* 12(8), pp. 459–476.

Quillian, 1966) Quillian, M. R.: *Semantic Memory*. Unpublished doctoral dissertation, Carnegie Institute of Technology, 1966. (Reprinted in part in M. Minsky (Ed.), *Semantic information processing*. Cambridge, Mass.: M.I.T. Press, 1968.)

Quillian, 1969) M. Ross Quillian: *The Teachable Language Comprehender: A Simulation Program in the Theory of Language*. *Commun. ACM* 12(8), pp. 459–476.

Quine and Ullian, 1978) Quine W. V., Ullian J.S.: *The Web of Belief*, Mc-Graw Hill, New York, 2nd edition.

Quine, 1990) W. V. Quine: *Pursuit of Truth*. Harvard Univ. Press, revised edition.

Quinlan, 1986) J. Ross Quinlan: *Induction of Decision Trees*. *Machine Learning*, Vol. 1, No. 1, pp. 1–33.

Quinlan, 1992) J. Ross Quinlan: *C4.5: Programs for Machine Learning*, Morgan Kaufmann Series in Machine Learning, Morgan Kaufmann.

Radhakrishnan, 1923) Radhakrishnan, S.: *Indian Philosophy, Vol. II*, current edition Oxford University Press, New Delhi, 2006.

Raiman, 1986) Olivier Raiman: *Order of Magnitude Reasoning. Proceedings of the 5th National Conference on Artificial Intelligence (AAAI 1986)*. Morgan Kaufmann, pp. 100–104.

Raiman, 1991) Olivier Raiman: *Order of Magnitude Reasoning. Artificial Intelligence*, Vol. 51(1–3), pp. 1–38.

Ramachandran, 2003) Ramachandran V., Ramachandran V.S., *The Emerging Mind*, Profile Books Ltd.

Ramachandran, 2010) Ramachandran V.S.: *The Tell-Tale Brain, Unlocking the Mystery of Human Nature*, Random House India.

Raman and Khemani, 1998) S. Shankara Raman, Deepak Khemani: "Synthesizing Solutions from Memory Chunks", AAAI Technical Report SS-98-04, *Proceedings of the 1998 AAAI Spring Symposium on Multimodal Reasoning*, AAAI Press, Menlo Park, pp. 102–105.

Raphael, 1976) Bertram Raphael: *The Thinking Computer: Mind Inside Matter (A series of books in psychology)*, W.H. Freeman & Co Ltd.

Reinelt, 2004) Reinelt G. TSPLIB. <http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/>, version on 26.11.2004

Reiter, 1980) Raymond Reiter: *A Logic for Default Reasoning. Artificial Intelligence*, 13, pp. 81–132.

Reiter, 1987) Raymod Reiter: *A Theory of Diagnosis From First Principles. Artificial Intelligence*, Vol. 32, pp. 57–97.

Rich and Knight, 1991) Elaine Rich, Kevin Knight: *Artificial Intelligence*, Tata McGraw Hill.

Rich, 1992) Elaine Rich: *Artificial Intelligence*, McGraw-Hill.

Petrie, 1992) Robertie, B: *Carbon versus Silicon: Matching wits with TD-Gammon*. In Blackgammon 2, 2, pp. 14–22.

Robinson and Wos, 1969) G.A. Robinson L. T. Wos: *Paramodulation and Theorem Proving in First Theories with Equality*. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, American Elsevier New York.

Robinson, 1965) J. Alan Robinson: *A Machine-Oriented Logic Based on the Resolution Principle*. *Journal of the ACM (JACM)*, Vol. 12, Issue 1, pp. 23–41.

Rohloff et al., 2007) Kurt Rohloff, Mike Dean, Ian Emmons, Dorene Ryder, and John Sumner: *Evaluation of Triple-Store Technologies for Large Data Stores*, in R. Meersman, Z. Tari, P. Herrero (Eds.): *OTM 2007 Ws, Part II, LNCS 4806*, Springer-Verlag Berlin Heidelberg, pp. 1105–1114.

Rose, 1998) Steven P. R. Rose: *How Brains Make Memories*. In Patricia Fara and Karalyn Patterson (Eds.), *Memory*, Cambridge University Press, Cambridge.

Rosenblatt, 1958) Rosenblatt, Frank: *The Perceptron: A Probabilistic Model for Information Storage Organization in the Brain*, Cornell Aeronautical Laboratory, *Psychological Review*, Vol. 65, No. 6, pp. 386–408.

Rosenblatt, 1962) Frank Rosenblatt: *Principles of Neurodynamics: Perceptrons and the Theory of Mechanisms*, Spartan Books, Washington DC.

Rosenkrantz et al., 1977) Rosenkrantz D.J., Stearns R.E., and Lewis P.M.: *An Analysis of Sequential Algorithms for the Traveling Salesman Problem*. *SIAM Journal of Computing*, Vol. 6, No. 3, pp. 311–331.

Rossi et al., 1990) Francesca Rossi, Charles J. Petrie, Vasant Dhar: *On the Equivalence of Consistency Satisfaction Problems*. In *Proc. European Conference on Artificial Intelligence (ECAI-90)*, Stockholm, pp. 550–556.

Rossi et al., 2006) Francesca Rossi, Peter van Beek, Toby Walsh (Eds): *Handbook of Consistency Management*, Elsevier.

Rostand, 1897) Edmond Rostand, Lowell Blair (Translator), *Cyrano de Bergerac*, 1897, available on Project Gutenberg Classics, 2003.

Rumelhart, 1986) Rumelhart, D.E., J.L. McClelland and the PDP Research Group: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Volume 1: Foundations, Cambridge MIT Press.

Rumelhart and Ortony, 1977) Rumelhart, D.E., and Ortony, A.: *The Representation of Knowledge in Memory*, in Eds., R.C. Anderson, R.J. Spiro, and W.E. Montague, *Schooling and the Acquisition of Knowledge*, Lawrence Erlbaum, Hillsdale.

Russell and Norvig, 2009) Stuart Russell and Peter Norvig: *Artificial Intelligence: A Modern Approach* (2nd edition), Prentice-Hall.

Russell, 1945) Bertrand Russell, *A History of Western Philosophy*, Routledge, 2 edition.

Sacerdoti, 1974) Earl D. Sacerdoti: *Planning in a Hierarchy of Abstraction Spaces*. *Artificial Intelligence Journal*, Vol. 5, No. 2, pp. 115–135.

Sacerdoti, 1977) Earl D. Sacerdoti: *A Structure for Plans and Behaviour*, Elsevier.

Samuel, 1959) Arthur, Samuel (1959). "Some Studies in Machine Learning Using the Game of Checkers". *IBM Journal* 3 (3), pp. 210–229.

Sartre, 1943) Jean-Paul Sartre: *Being and Nothingness: An Essay on Phenomenological Ontology*, Routledge; second edition, 2003.

Schaaf, 1995) Schaaf, J. W., Fish and Sink: *An Anytime-Algorithm to Retrieve Adequate Cases*. In Amodeo, A. and M. Veloso (Eds.) (1995). *Case-Based Reasoning Research and Development, CBR-95, Lecture Notes in Artificial Intelligence, 1010*. Springer Verlag., (1995), pp. 538–547.

Schaaf, 1996) Schaaf, J. W.: *Fish and Shrink: A Next Step Towards Efficient Case Retrieval in Large Scaled Case Bases*. In Smith, I. and B. Faltings (Eds.) (1996). *Advances in Case-Based Reasoning, Lecture Notes in Artificial Intelligence, 1186*. Springer-Verlag.: pp. 362–376.

Schaeffer et al., 1992) Jonathan Schaeffer, Joseph Culberson, Norman Treloar, Brent Knight, Paul Hsieh and Duane Szafron: *A World Championship Caliber Checkers Program*, *Artificial Intelligence*, Vol. 53, pp. 2–3, pp. 273–290.

Schaeffer, 1997) Jonathan Schaeffer. *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer-Verlag.

Schank and Abelson, 1977) Schank, Roger C., Robert P. Abelson: *Scripts, Plans, Goals, and Understanding: An Inquiry into Human Knowledge Structures*. Hillsdale, NJ: Lawrence Erlbaum.

chank and Colby, 1973)

ank et al., 1975) Roger C. Schank, Neil M. Goldman, Charles J. Rieger III, Christopher Rieser: *Reference and Paraphrase by Computer*. J. ACM 22(3), pp. 309-328.

ank, 1982) Roger C. Schank: *Dynamic Memory: A Theory of Reminding and Learning in Computers and People*. Cambridge University Press.

ank, 1999) Roger C. Schank: *Dynamic Memory Revisited*, Cambridge University Press.

midt-Schauß and Smolka, 1991) Manfred Schmidt-Schauß and Gert Smolka: *Attributive Concept Descriptions with Complements*, *Artificial Intelligence*, Vol. 48, No. 1, pp. 1–26.

roeder, 1997) Michael Schroeder: *A Brief History of the Notation of Boole's Algebra*, *Nordic Journal of Philosophical Logic*, Vol. 2, No. 1, pp. 41–62.

available at <http://www.hf.uio.no/ifikk/filosofi/njpl/vol2no1/history/>

ubert, 1990) L.K. Schubert: *Monotonic Solution of the Frame Problem in the Situation Calculus: An Efficient Method for Worlds with Fully Specified Actions*. In H. Kyburg, R. Loui and G. Carlson (eds): *Knowledge Representation and Defeasible Reasoning*, Kluwer, Dordrecht, pp. 23–67.

le, 1980) John Searle. *Minds, Brains, and Programs*, *Behavioral and Brain Sciences* 3, pp. 417–457.

amani and Khemani, 2003) Radhika Selvamani, B and Deepak Khemani: *Managing Experience for Process Improvement in Manufacturing*. In Ashley K.D., and Bridge D.G. (eds) : *Case-Based Reasoning Research and Development, Proceedings of 5th Int. Conf. on Case-Based Reasoning, ICCBR-03*, Trondheim, Norway, June 2003, Springer LNAI 2689, pp. 522–536.

mour and Norwood, 1993) Jane Seymour and David Norwood, 1993: "A Game for Life," *New Scientist* 10. 1889, 04 September 1993, pp. 23–26.

er, 1976) Glenn Shafer: *A Mathematical Theory of Evidence*. Princeton University Press. (Shanahan, 1995) Murray Shanahan: *A Circumscriptive Calculus of Events*. *Artif. Intell.* 77(2), pp. 249-284.

anahan, 1997) M.P.Shanahan: *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*, MIT Press.

anahan, 1999) Murray Shanahan: *The Event Calculus Explained. Artificial Intelligence Today 1999*, pp. 399-430.

annon, 1948) Claude Shannon: *A Mathematical Theory of Computation*. *Bell System Technical Journal*, Vol. 27, No. 3, pp. 379–423.

annon, 1950) Shannon, Claude E. (1950), "Programming a Computer for Playing Chess," *Philosophical Magazine Ser.7*, Vol. 41(314).

piro and Rapaport, 1986) Stuart C Shapiro and William J. Rapaport: *SNePS Considered as a Dimensional Propositional Semantic Network.: Proceedings of the 5th National Conference on Artificial Intelligence*. Philadelphia, PA, Morgan Kaufmann, pp. 278–283.

piro, 2000) Stuart C Shapiro: *An Introduction to SNePS 3*. In Bernhard Ganter and Guy W. M. Leuzinger (eds.), *Conceptual Structures: Logical, Linguistic, and Computational Issues, Lecture Notes in Artificial Intelligence 1867*, Springer-Verlag, pp. 510–524.

rma, 2003) Rama Kant Sharma: *Hardy and The Rasa Theory*, Sarup & Sons.

ley, 2001) Mary Shelley: *Frankenstein*, Oxford University Press, USA; New edition.

ppard, 2002) Brian Sheppard: *World-championship-caliber Scrabble*, *Artificial Intelligence* 134 (2002), pp. 241–275.

rtcliffe, 1976) Shortliffe E.H.: *Computer Based Medical Consultation: MYCIN*, Elsevier, New York.

on, 1969) Simon, H. A.: *Sciences of the Artificial*. Cambridge, MA: MIT Press.

on, 1974) Herbert A. Simon: *The Sciences of the Artificial Intelligence*, MIT Press, Cambridge M.A.

nton, 2004) Dean Keith Simonton: *Creativity in Science: Chance, Logic, Genius, and Zeitgeist*, Cambridge University Press.

s, 94) Karl Sims: *Evolving Virtual Creatures*. In Andrew Glassner, editor, *Proceedings of SIGGRAPH 94* (Orlando, Florida, July 24-29).

a and Vidyabhusana, 1930) Nandalal Sinha: Mahamahopadhyaya Satisa Chandra Vidyabhusana: *The Nyaya Sutras of Gotama, The Sacred Books of the Hindus*, Motilal Banarsidass, 1930. Munshi Manoharlal reprint, 2003.

le, 1961) Slagle: J. *A Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculus*. Ph.D. diss., MIT, May 1961.

le, 1963) Slagle, James R.: "A Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculus", *JACM*, Vol. 10, No. 4, October 1963, pp. 507–520.

h and Weld, 1998) David E. Smith, Daniel S. Weld: *Conformant Graphplan*, *Proceedings of AAAI-98*, AAAI Press, Menlo Park, CA, 1998, pp. 154–159.

h, 2003) Barry Smith, Ontology, in (ed.) Luciano Floridi: *Blackwell Guide to Philosophy of Science and Information*, Wiley-Blackwell.

h, 2007) Robin Smith, *Aristotle's Logic*, First published Mar 18, 2000: Substantive revision Dec 2007. In Edward N. Zalta (ed.), *Stanford Encyclopedia of Philosophy*.
available at <http://plato.stanford.edu/entries/aristotle-logic/>

llyan, 1979) Smullyan, Raymond: *The Chess Mysteries of Sherlock Holmes*. New York: Knopf, 1979.

llyan, 1992) Smullyan, Raymond: *The Chess Mysteries of the Arabian Knights*. Oxford Oxford University Press.

llyan, 1992) Raymond M. Smullyan: *Godel's Incompleteness Theorems*, Oxford University Press.

th and McClave, 2001) Smyth, B. and McClave, P. 2001. *Similarity vs. Diversity*. In *Proceedings of the 4th international Conference on Case-Based Reasoning: Case-Based Reasoning Research and Development* D. W. Aha and I. Watson, Eds. *Lecture Notes In Computer Science*, Vol. 2080. Springer-Verlag, pp. 347–361.

ayajulu, 1998) S.G.Somayajulu: *Communicating Plans in Natural Language: Planning and Realization*, Ph.D. thesis, IIT Madras.

a, 1984) John F. Sowa: *Conceptual Structures: Information Processing in Mind and Machine*. Addison-Wesley Publishing Company, Reading Massachusetts.

a, 2000) John F. Sowa: *Knowledge Representation: Logical, Philosophical, and Computational Foundations*, Brooks/Cole, Thomson Learning.

a, 2006) John F. Sowa: *Semantic Networks*, at <http://www.jfsowa.com/pubs/semnet.htm>

a, 2009) John F. Sowa: *Conceptual Graphs for Representing Conceptual Structures*. In Peter Hitzler and Henrik Scharfe (Eds.), *Conceptual Structures in Practice*, Chapman & Hall/CRC.

re, 1987) Larry R., Squire. *Memory and Brain*. Oxford University Press, New York.

re, 2004) Larry R., Squire. *Memory Systems of the Brain: A Brief History and Current Perspectives*. *Neurobiology of Learning and Memory*, 82, pp. 171–177.

re, 2007) Larry R. Squire: *Learning and Memory*. In, Floyd E. Bloom, M. Flint Beal, and David S. Saper, (Eds.), *The Dana Guide to Brain Health, A Practical Family Reference from Medical Experts*. Dana Press, 2006. Updated 2007 article <http://www.dana.org/news/brainhealth/detail.asp?ID=10020>

n, 1989) Lynn Andrea Stein: *Skeptical Inheritance: Computing the Intersection of Credibility Extensions*. In N. S. Sridharan: *Proceedings of the 11th International Joint Conference on Artificial Intelligence, IJCAI*, pp. 1153–1160.

n, 1992) Lynn Andrea Stein: *Resolving Ambiguity in Nonmonotonic Inheritance Hierarchies*. *Artificial Intelligence* 55(2), pp. 259–310.

ing and Nygate, 1990) Leon Sterling, Yossi Nygate: *PYTHON: An Expert Squeezer*. *J. Log. Programming*, pp. 21–39.

ing and Shapiro, 1994) Leon Sterling, Ehud Shapiro, *The Art of Prolog*, Second Edition: *Advanced Programming Techniques (Logic Programming)*, The MIT Press.

art, 1998) Ian Stewart: *Mathematical Recreations: Monks, Blobs, and Common Knowledge*. *Scientific American*, Vol. 279, No. 2, Aug 1998, pp. 96-97.

art, 2002) Ian Stewart: *Does God Play Dice? The New Mathematics of Chaos*, Wiley-Blackwell, second edition.

ckman, 1979) George C. Stockman: *A Minimax Algorithm Better than Alpha-Beta?* *Artif. Intell.* 13(2), pp. 179-196.

ng, 2009) Strang, G.: *Introduction to Linear Algebra* Wellesley-Cambridge Press, Fourth Edition.

ss and Dressler, 1989) Peter Struss, Oskar Dressler: "Physical Negation" Integrating Fault Models into the General Diagnostic Engine. In N. S. Sridharan (ed.), *Proceedings of the 11th International Conference on Artificial Intelligence, IJCAI 1989*, Morgan Kaufmann, pp. 1318–1323.

ss and Price, 2004) Peter Struss, Chris Price: *Model-Based Systems in the Automotive Industry*. *Magazine*, Vol. 24(4), pp. 17–34.

ss, 2008) Peter Struss: *Model Based Problem Solving*. In F. van Harmelen, V. Lifschitz and B. Bonet (Eds.), *Handbook of Knowledge Representation*, Elsevier.

ptner and Wotawa, 1998) Stumptner M, Wotawa F.: *Model-based Reconfiguration*. In Gero, J. and Sudweeks, F. (Eds), *Proceedings of the Fifth International Conference on Artificial Intelligence Design (AID'98)*, Kluwer Academic Publishers.

1975) Spearman, C. A. *Correlation Models of Ability Acquisition*. *Flourish* (Northampton)

23. (Tesauro, 1989) Tesauro, G.: *Neurogammon wins Computer Olympiad*. *Neural Computation* 1, pp. 21–33.
24. (Tesauro, 1994) G. Tesauro (1994): “*TD-Gammon, A Self-Teaching Backgammon Program Achieving Master-level Play*” in *Neural Computation*, Vol. 6, No 2, pp. 215–19.
25. (Tesauro, 1995) Gerald Tesauro: *Temporal Difference Learning and TD-Gammon*. *Commun. ACM* 38(1), pp. 58–68.
26. (Tesauro, 2002) Gerald Tesauro: *Programming Backgammon using Self-teaching Neural Nets*. *Artif. Intell.* 144(1–2), pp. 181–199.
27. (Throop, 1983) Thomas A Throop: *Computer Bridge*, Hayden Book Co.
28. (van Heer and Janko, 1999) Joost de Heer and Otto Janko: *The Retrograde Analysis Corner*. <http://www.janko.at/Retros/>
29. (Tsang, 1993) Edward Tsang, *Foundations of Constraint Satisfaction*, Academic Press.
30. (Turing, 1936) A. M. Turing: *On Computable Numbers, with an Application to the Entscheidungs Problem*. *Proceedings of the London Mathematical Society*, ser. 2, Vol. 42, No. 1, DOI:10.1112/plms/s2-42.1.230 pp. 230–65.
31. (Turing, 1937) A. M. Turing, A.M.: *On Computable Numbers, with an Application to the Entscheidungs Problem: A correction*, *Proceedings of the London Mathematical Society*, ser. 2, Vol. 43, No. 1, DOI:10.1112/plms/s2-43.6.544 pp. 544–6, 1937.
32. (Turing, 1950) A. M. Turing: *Computing Machinery and Intelligence*, *Mind* 59, DOI:10.1093/mind/LIX.236.448 <http://loebner.net/Prize/TuringArticle.html>, pp. 433–460.
33. (Uexküll, 1982) Uexküll, J. V. (1982 (1940)): *The Theory of Meaning*, *Semiotica* 42 (1), pp. 25–82.
34. (Vaidya et al., 2009) Ashwini Vaidya, Samar Husain, Prashanth Mannem, and Dipti Misra Sharma: *Araka Based Annotation Scheme for English*, in A. Gelbukh (Ed.): *CICLing 2009*, Springer-Verlag, LNCS 5449, pp. 41–52.
35. (Van Beek and Chen, 1999) Peter van Beek, Xinguang Chen: *CPlan: A Constraint Programming Approach to Planning*. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Innovative Applications of Artificial Intelligence Conference*, American Association for Artificial Intelligence, Menlo Park, CA, pp. 585–590.
36. (Van Berendonck and Jacobs, 2003) Christopher Van Berendonck, Timothy Jacobs: *Bubbleworld: A Visual Information Retrieval Technique*. In Tim Pattison, Bruce H. Thomas (Eds.): *Australian Symposium on Information Visualization*, InVis.au, Adelaide, Australia, 2003. CRPIT 24 Australian Computer Society 2003, pp. 47–56.
37. (Van Dalen, 2001) Dirk Van Dalen: *Intuitionistic Logic*, in Goble, Lou, ed., *The Blackwell Guide to Philosophical Logic*. Blackwell.
38. (Van Heijenoort, 1967). J. vanHeijenoort (ed.): *From Frege to Gödel: A Source Book in Mathematical Logic*, Cambridge, MA: Harvard University Press.
39. (Van Hentenryck, 1989) Van Hentenryck, P.: *Constraint Satisfaction in Logic Programming*, Cambridge, MA: MIT Press.
40. (Vasu, 1962) Srisa Chandra Vasu: *The Ashtadhyayi of Panini*—2 Vols., Motilal Banarsidass.
41. (Vatsayayan, 1996) Kapila Vatsayayan: *Bharata: The Nāṭyaśāstra*, Sahitya Academy, New Delhi, 1996.
42. (Veloso and Carbonell, 1993) Veloso, M.M. & Carbonell, J.G.: *Planning and Learning by Analogical Reasoning*. *Machine Learning*, Vol. 10.
43. (Vidyabhusana, 1978) Satish Chandra Vidyabhusana: *A History of Indian Logic*, Motilal Banarsidass, New Delhi, reprinted 1988.
44. (Vidyabhusana, 2003) Satis Chandra Vidyabhusana: *Nyaya Sutras of Gotama*, Munshiram Manoharlal, New Delhi.
45. (Wallace and Freuder, 1992) Wallace, R. J. and E. Freuder: *Ordering Heuristics for Arc Consistency Algorithms*, *Proc. Ninth Canad. Conf. on AI*, Vancouver, pp. 163–169.
46. (Wallace, 1993) Richard J. Wallace: *Why AC-3 is Almost Always Better than AC4 for Establishing Consistency in CSPs*. In Ruzena Bajcsy (Ed.): *Proceedings of the 13th International Joint Conference on Artificial Intelligence*. Chambéry, France, August 28 -September 3, 1993. Morgan Kaufmann, pp. 239–247.
47. (Valtchev and Missouri, 2001) Petko Valtchev and Rokia Missaoui: *Building Concept (Galois) Lattices from Parts: Generalizing the Incremental Methods*, in Harry S. Delugach, Gerd Stumme: *Conceptual Structures: Broadening the Base*, 9th International Conference on Conceptual Structures, ICCS 2001, LNCS 2120, Springer.
48. (Waltz and Barlow, 1985) David L. Waltz and Jordan B. Barlow: *Massively Parallel Processing: A Study of*

on, 2002) Ian Watson: *Applying Knowledge Management: Techniques for Building Corpora* (The Morgan Kaufmann Series in Artificial Intelligence), Morgan Kaufmann.
 enbaum, 1966) Joseph Weizenbaum: *ELIZA—A Computer Program For the Study of Natural Language Communication Between Man And Machine*, *Communications of the ACM* 9 (1), pp. 36–45.
 et al., 1998) Daniel S. Weld, Corin R. Anderson, David E. Smith: *Extending Graphplan to Handle Uncertainty and Sensing Actions*. *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98*, AAAI Press / The MIT Press, pp. 897–904.
 d, 1994) Daniel S. Weld: *An Introduction to Least Commitment Planning*. *AI Magazine* 15(4), pp. 3–10.
 os, 1994) P. J. Werbos: *The Roots of Backpropagation: From Ordered Derivatives to Networks and Political Forecasting*, Wiley.
 ehead and Russell, 1910-13) Alfred North Whitehead and Bertrand Russell: *Principia Mathematica*, Cambridge University Press, 1910, 1912, and 1913. Second edition, 1925 (Vol. 1), 1927 (Vol. 2).
 rf, 1956) Whorf, B.L.: "The Relation of Habitual Thought and Behavior to Language." In J.B. Cowley (ed.) *Language, Thought, and Reality: Selected Writings of Benjamin Lee Whorf* (pp.134–149). Cambridge, MA: MIT.
 nsky, 1981) Robert Wilensky: *PAM*. In (Schank and Riesbeck, 1981).
 nsky, 1983) Robert Wilensky: *Planning and Understanding: A Computational Approach to Human Reasoning*, Addison-Wesley, Longman Publishing Co.
 s, 1995) Andrew Wiles: *Modular Elliptic Curves and Fermat's Last Theorem*, *Annals of Mathematics* 141 (3), pp. 443–551.
 1988) Winkins D.: *Practical Planning: Extending the Classical AI Planning Paradigm*, Morgan Kaufmann.
 e, 1982) Wille, R.: *Restructuring Lattice Theory: An Approach Based on Hierarchies of Concepts*, I. (ed.) *Ordered Sets*. 445–470. Dordrecht-Boston, Reidel.
 e, 2005) *Formal Concept Analysis as Mathematical Theory of Concepts and Concept Hierarchies*, Bernhard Ganter, Gerd Stumme, Rudolf Wille: *Formal Concept Analysis, Foundations and Applications*, Springer.
 ams and Nayak, 1996) Brian C. Williams and P. Pandurang Nayak: *A Model-based Approach to Reactive Self-configuring Systems*. In *Procs. of AAAI-96*, AAAI Press, pp. 971–978.
 ams and Nayak, 1997) Williams, B. C., and Nayak, P. P.: *A Reactive Planner for a Model-based Executive*. In Pollack, M. E. (Ed.), *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, Morgan Kaufmann, pp. 1178–1195.
 ograd, 1971) Terry Winograd: *Procedures as a Representation for Data in a Computer Program for Understanding Natural Language*, MIT AI Technical Report 235, February 1971 Reuters. (1971) Reuters-21578 Text Classification corpus. Retrieved from daviddlewis.com/resources/collections/reuters21578/
 ston, 1977) Patrick Henry Winston: *Artificial Intelligence*. Addison-Wesley Pub. Co.
 ston, 1992) Patrick Henry Winston: *Artificial Intelligence* (3rd Edition). Addison-Wesley Pub. Co.
 genstein, 1921) Ludwig Wittgenstein: *Tractatus Logico-Philosophicus (Logisch-Philosophische Abhandlung*, *Annalen der Naturphilosophie* 1921), translation reprinted, Routledge, 2001.
 ff, 1993) Karl Erich Wolff: *A First Course in Formal Concept Analysis—How to Understand Formal Diagrams*, in, Faulbaum, F. (ed.) *SoftStat'93 Advances in Statistical Software* 4, 429–438, Springer-Verlag, available online at http://www.fbmn.fhdarmstadt.de/home/wolff/Publikationen/A_First_Course_in_Formal_Concept_Analysis
 erton and Hayes-Roth, 1994) Michael Wolverton and Barbara Hayes-Roth: *Retrieving Semantic Analogies with Knowledge-Directed Spreading Activation*. *Proceedings of the 12th National Conference on Artificial Intelligence*, Volume 1, AAAI 1994, pp. 56–61.
 1900) Michael Woolridge: *Reasoning about Rational Agents*. MIT Press, Cambridge, Massachusetts.
 lsey, 2000) Kit Woolsey: *Computers and Rollouts*, *GammOnline*, available at <http://gammonline.com/members/Jan00/articles/roll.htm>
 and Palmer, 1994) Wu, Z., and Palmer, M. 1994: *Verb Semantics and Lexical Selection*. In *Annual Meeting of the Association for Computational Linguistics*, pp. 133–138.
 (1966) Frances A. Yates: *The Art of Memory*. University of Chicago Press, 1966. Reprint edition, 1992.

Index

Symbols

1-consistent 290

3-consistent 299

A

a cutoff 249

A* algorithm 128, 365

abduction 439, 772

abductive inferences 439

abnormality predicates 712

ABox 588 abstract concept 506

abstraction 551, 552

abstraction hierarchy 545

abstractive summarization 704

abstract objects 551

ABSTRIPS 223

AC-1 293

AC-3 293

AC-4

algorithm 294

accommodation 549

ACE (Affect as a Consequence of Empathy) 580

actions 192, 510, 570

action schemas 416

activation 668

activation function 107

activation spreading 559, 669

adaptation 673

adaptive consistency 309

adjacency representation 99

adjusted sum-action heuristic 383

adjusted sum-duration heuristic 383

admissibility 133

admissible 130, 138, 147

admissible path 582

AdvanceTime 381

- agent 27, 410, 625
- aggregation function 644
- aggregation hierarchy 545
- a kind of 552 AL 600
- Allen's interval algebra 375
- AlphaBeta algorithm 248, 251
- alpha nodes 181
- Alternating Edges Crossover 100
- ambiguity 681, 686
- analytic reasoning 674
- anaphora resolution 684, 693
- AND arc 158
- AND node 158
- AND/OR problem 158, 242
- AND/OR trees/graphs 158
- anonymous individuals 597
- Answer predicate 465
- Ant Colony Optimization 112
- antecedents 438
- antitone Galois connection 608
- antonymy 687
- anytime algorithm 150
- AO 158
- AO* algorithm 163
- arc consistency 291, 293
- Arc Consistency Lookahead 319
- arc consistent 291
- arguments 421
- artificial life 109
- artificial neural networks (ANN) 847
- assertional reasoning 588
- assimilation 549
- association rule mining 865
- associative memories 863
- assumption based reasoning 752
- Assumption Based Truth Maintenance System (ATMS) 755
- assumptions 744
- at-end conditions 377
- at-end effect 377
- ATMS 757
- atomic concepts 585
- atomic sentence 435
- at-start condition 378
- at-start effects 377
- autoepistemic logic 717
- autonomous systems 410
- autonomous underwater vehicles 397

axon 106

B

- b cutoff 249
- BABEL 565
- Backgammon 276
- background knowledge 697
- BackJumping algorithm 322
- Backpropagation algorithm 855, 858
- Backtracking 312
- Backtracking-with-LookAhead 317
- backward chaining 169, 476, 477
- backward reasoning 156, 169, 218
- Backward State Space Planning 199
- Ball Overlaps Bounds 658
- Ball Within Bounds (BWB) 658
- basin of attraction 862
- Bayesian belief networks 775
- Bayesian estimation 810
- Bayesian network 814
- Bayesian probabilities 763
- Bayesian Reasoning 766, 772
- Bayes' rule 770
- BBNs 775, 777
- Beam Search 70, 148
- Beam Stack Search (BSS) 148
- beam width 70
- belief 773, 799
- Belief-Desire-Intention 410
- belief networks 775
- beliefs 411, 716, 745, 796
- Bellman backup 790
- Bellman update 790
- best first search 56, 128
- beta node 181
- binary connectives 435
- binding constraints 212
- biological neuron 106
- biosemiotics 103
- blind 50
- Blocks World Domain 194, 196
- Board Games 237
- BOB Test 658
- Boltzmann machine 862
- Boolean attributes 637
- Boolean functions 852

- BORIS 577
- bottom-up filtering 691
- bottom-up parser 690, 691
- boundary 145
- Bounded Greedy (BG) 671
- Branch & Bound 120
- Breadth First Heuristic Search (BFHS) 147
- Breadth First Search (BFS) 39
- broken 748, 750
- B* Search 261
- bucket 655, 658
- building taxonomies 592
- BWB Test 658

C

- C4.5 836
- CAIP 399
- calculus ratiocinator 427
- Candidate-Elimination 827
- candidate generation 741
- candidate space 742
- case 626
- Case Based Reasoning (CBR) 628
- case cohesion 651
- case completion 636
- case markers 536
- Case Retrieval Net (CRN) 668
- categories 430, 491, 505, 545
- causal graphs (CG) 372
- causal link 210, 361
- causal relation 775, 815
- CBR methodology 629
- CD ACTs 528
- CD theory 522
- cellular automaton 109
- centroid 838
- certainty factor 773
- change 492
- characteristica universalis 585
- Checkers 242
- CHEF 676
- Chess 242
- Chinese room 426
- chromosome 87
- chronological backtracking 156, 314
- chunking 550, 565
- chunks 628

circumscribed 713
Circumscription 518, 712, 713
city block distance 55, 653
classical planning 193
classification 592, 674, 769, 851
classifier 848
classify patterns 862
clause form 461
clauses 444
CLAVIER 676
clobbered 212
CLOSED 34, 131
closed world assumption 710
cluster 838
cluster assignments 839
cognitive schema 549
colony of ants 112
CombEx 33
comparable 726
competent GAs 100
competitive layer 861
complete 138
complete information games 238
completeness 40, 57, 66, 424
complexity 470
component behaviour 729
component library 737
compositional adaptation 674
compositional semantics 683
compound label 290
Computer Bridge 269
concept based information retrieval 696
concept forming 585
concept lattice 609
concept learning 823
concepts 491, 548, 552, 587
conceptual actions 525
Conceptual Analysis 532
Conceptual Dependency Theory 522, 564
Conceptual Graphs 613
conceptualizations 522, 524, 528
conditional dependence 775
conditional effects 344
conditional independencies 815
conditionally independent 776
conditional probabilities 766
confidence interval 773, 799, 801

- configuration 170, 400
- configuration problems 36
- configuration rule 401
- Conflict Directed Backjumping 326
- conflict recognition 741
- conflict resolution 182
- conflict resolution strategies 176
- Conflict Set (CS) 175, 179, 740
- confluences 727, 728
- Conformant Graphplan (CGP) 347, 348
- conformant planning 348
- CONGEN 160
- conjunctive normal form (CNF) 68, 444
- connectionist network 562
- consequent 438
- consistency 424, 440
- consistency based diagnosis 736, 738
- consistency establishment 290
- consistency property 137
- consistent belief 761
- constant symbols 449
- constraint network 286, 298
- constraint propagation 290, 299, 303, 729
- constraints 286, 354, 400, 730, 732, 745
- constraint satisfaction problems 286, 328
- constructive adaptation 677
- content 438
- content addressable memories 636, 863
- contingent planning 347
- Contract Bridge 266
- contradictory assumptions 755
- conversational CBR 632, 634
- coreference analysis 702
- Corpus Based MT 703
- corresponding values 731, 733
- creativity 100
- creature 110, 412
- credit assignment 278, 843
- credulous extension 583
- credulous reasoner 584, 717
- CRIKEY 384
- CRIKEY3 384
- crossover 89, 97
- crossword puzzle 329
- cryptarithmic problem 329
- CSP 286
- culprit variable 322

- cut-fail combination 481
- cutoffs 250
- cut operator 480
- CWA 710
- cycle crossover 98

D

- DAC 306
- Dartmouth Conference 234
- data-driven 168
- data-driven reasoning 169
- data mining 416, 809
- dead-end variable 322
- debating 432
- decision trees 832
- declarative knowledge 416
- declarative memory 414
- declarative model 736
- declobbered 212
- deduction 430
- deductive retrieval 465
- deductive system 439
- Deep Blue 234, 244
- deep knowledge 625, 736
- Deep Thought 234, 265
- default behaviour 739
- default logic 429, 715
- default reasoning 709
- default rules 175, 715
- default theory 715
- deferred heuristic evaluation 373
- definite 472
- definite clauses 477
- degrees of belief 763
- delta rule 851
- De Morgan's law 452
- demotion 213
- Dempster-Shafer (D-S) theory 797
- dendral 160, 185
- dendrites 106
- dendritic tree 106
- dense solution path 146
- dependency directed backtracking 156, 322, 752
- Depth Bounded DFS (DBDFS) 47
- Depth First Iterative Deepening (DFID) 47
- Depth First Search (DFS) 38

- derivation 463, 464, 683
- derivational adaptation 673, 677
- description logic (DL) 412, 585, 600
- device behaviour 729
- device structure 729
- DFID 48
- diagnosis 736, 740, 772
- difficulty heuristic 367
- Dijkstra's algorithm 127
- DIMACS implementation challenge 93
- dimensionality reduction 699
- directed acyclic word graph (DAWG) 272
- direct experience 626
- directional arc consistency (DAC) 306
- directional arc consistency lookahead 319
- directional consistency 305
- directional path consistency (DPC) 307
- direct MT 702
- discounting factor 780
- discrimination trees 832
- disjunctive facts 461
- distance 637
- distance, inter-quartile 657
- diversity 631
- Diversity Conscious Retrieval (DCR) 670, 672
- Divide and Conquer Beam Search (DCBS) 148
- Divide and Conquer Beam Stack Search (DCBSS) 150
- Divide and Conquer BFHS (DCBFHS) 148
- Divide and Conquer Frontier Search (DCFS) 144
- DL language 585
- DMAP (Direct Memory Access Parsing) 568
- domain 450
- domain closure 711
- domain experts 169, 171
- domain independent heuristics 328
- domain knowledge 119
- domain transition graph (DTG) 371
- double dummy problem 269
- driver log domain 232
- d-separation 777
- DTG 373
- Dublin Core Metadata Initiative 498
- durative actions 374, 375

E

- edit distance 685

effective branching factor 59
effective rank 700
effects-(a) 198
effects(a) 198
effects+(a) 198
eigenvalue 698
eigenvectors 698
Eight Puzzle 31, 55
Elementa Calculi 427
ELI 535
emergent systems 103, 108, 112
Empirical methods in NLP 684
empty plan 209
Enforced Hill Climbing (EHC) 368
English Language Interpreter (ELI) 565
entailment 421, 437, 710
entropy 833
environment 744, 757
envision 728
envisionment 723
episodes 626
episodic knowledge 413
episodic memory 567
equality statement 467
error correcting rule 848
estimated solutions 320
Euclidean distance 55, 653
Euclidean TSP 93
EUROPA 399
evaluation function 88, 243, 845
Event Calculus (EC) 510, 511
events 510
evidence 796
evidential reasoning 797
evolutionary algorithms 112
excitatory 562
existential goals 465
existential graphs 614
existentially quantified variables 458
existential quantifier 448, 451
expansion 718
Expectation Maximization (EM) 693, 704, 839
expectations 540, 564
expected utility 780, 783
experience 409, 625
experiential knowledge 628, 736
expert systems 169, 171, 185

- explaining away 774
- explicit semantic analysis (ESA) 697
- exploitation 82
- exploration 82
- extensions 492, 583, 607, 716
- extractive summarization 704

F

- factor analysis 697
- facts 451
- false negative 675
- false positive 675
- Fast Downward 370
- Fast Forward (FF) 367
- fault localization 737
- fault models 747
- fault modes 750, 751, 760
- feedforward network 854
- fighting value 244
- Find-S 825
- finite domains 287
- finite horizon MDPs 780
- Finite State Transducers 686
- First Order Logic (FOL) 425, 428, 448
- Fish and Shrink Algorithm 663
- fitness function 88
- five step syllogism 433
- flaws 212, 213
- fluents 195, 358, 510
- $f(n)$ 128
- $f^*(n)$ 128
- $f(\text{node})$ 128
- focused iterative broadening search 374
- FOL Rules 452
- FOL Semantics 449
- FOL Syntax 448
- form 438
- Formal Concept Analysis 606
- formal context 606
- formalization 428
- formal logic 416, 420, 427, 438
- FormTool 676
- formula 424, 435
- forward chaining 169, 170
- forward chaining in FOL 453
- Forward Checking 317

- forward reasoning 441, 169
- Forward State Space Planning 197
- frame axioms 357, 195
- frame based representations 413
- frame of discernment 797
- frame problem 195, 517
- frames 550
- frame system 554, 556
- Friend Of A Friend (FOAF) 501
- Full Lookahead 319
- function symbols 449
- fuzzy sets 508

G

- GADDAG 274
- Game of Life 109
- game tree 238
- gaps 570, 572
- garden path sentences 538
- general boundary 827
- General Diagnostic Engine (GDE) 741
- generalization 452
- Generalized Closed World Assumption (GCWA) 711
- General Problem Solver 219
- Generate And Test() 29
- Genetic Algorithms 88
- genetic representation 110
- genome 88
- genotype 87, 110
- global similarity 636, 643
- $g(n)$ 128, 130
- $g^*(n)$ 130
- Go 242, 265
- goal competition 575
- goal concord 575
- goal conflict 575
- goal description 388
- goal directed 169
- goals 411, 570
- Goal Stack Planning 204
- goal states 27, 780
- goal subsumption 575
- goalTest 29
- goal trees 158, 161, 479
- Goban 265
- GP-CSP 353

- gradient 851
- gradient descent 859
- gradient descent rule 851
- GraphBackjumping 324
- graph databases 498, 501
- Graphplan 331, 340
- graph search algorithm 131
- greedy heuristic 94
- gripper domain 231

H

- Hamming distance 638
- Heaviside function 108
- Hebbian learning 860
- helpful actions 369
- heuristic crossover 99
- heuristic estimate 163
- heuristic functions 53, 119
- heuristic knowledge 416
- heuristic search 53, 362
- Heuristic Search Planner 363
- heuristic value 130
- hidden layer 854
- Hidden Markov Model (HMM) 689, 816
- hidden neurons 854
- hierarchical planning 225
- hierarchies 547
- high level plan 393
- Hill Climbing 61, 862
- history 425
- hitting set 740
- HMM Training 818
- $h(n)$ 130
- $h^*(n)$ 130
- holonymy 687
- homographs 539
- homunculus 426
- Hopfield network 862
- Horn clauses 472
- HSP 363
- HSP2 365
- HSPr 365
- HTN planning 228
- Huffman-Clowes scene labelling problem 296
- hypernymy 687
- hyponym 641

hyponymy 687
hypotheses space 824
hypothesis 823

I

IC-1 302
i-consistency 302
ID3 (Iterative Dichotomiser 3) 832
IDA* 139
idealism 105
ideally skeptical reasoning 584
ideas 427
imagine 420, 491
implicit quantifier form 454, 455
inadmissible 148
incompleteness 460
Incompleteness Theorem 483
indefinite horizon MDPs 780
independent actions 376
induced graph 306
induced mutex 351
induced width 306
inductive bias 831
inductive learning 823, 832
inertia axiom 513, 516
inference 422
inference engine 169, 171, 175
inferencing and event merging 702
infinite horizon MDPs 780
inflection 683
information extraction 701
information gain 657, 833
information processing 413
information retrieval 693
informed search 60
inheritance 556, 580, 588
inheritance graph 582
inheritance network 581
inhibitory 562
Inreca 662
Instance-Of 552, 587
intensions 492, 607
intentions 411
interlingua 702
internal dead end 323
interpretation 451, 713

- interpretation function 450
- inter quartile distance 657
- introspective knowledge 697
- Intuitionistic Logic 429
- inverse document frequency (idf) 695
- inverted file 696
- IP2 346
- IR 693
- IR system 694
- Is-A 552, 587
- is-a-kind-of 559
- Iterative Deepening 139
- Iterative Policy Evaluation 786

J

- joint probability distribution 767
- justification 715, 745, 752

K

- kd-tree 654
- kind of stuff 541
- K-means Clustering 838
- K Nearest Neighbour (KNN) 630
- KNN retrieval 652, 658
- knowledge 170, 409, 433, 627
- knowledge and belief 521
- knowledge base 711
- knowledge based 625
- knowledge based reasoning 407
- knowledge based systems 186
- knowledge discovery 416
- knowledge representation 412, 416, 544, 545
- knowledge structures 558
- Kohonen Network 861
- Kripke semantics 429

L

- label sequence 816
- landmarks 732
- landmark values 723, 730
- LAO* 792
- Latent Semantic Indexing 697
- leaf dead end 323
- learning 808, 847

- learning rate 846, 848
- least commitment strategy 212, 224
- least estimated cost 124
- Levenshtein distance 638
- lexical analysis 702
- lexical semantics 682
- lexicon 535
- linear association classifier 854
- linear combination 699
- linear discriminator 850
- linearly separable 850
- linear plan 217
- linear planning 204
- linguistic knowledge 697
- LMS update 846
- local maxima 62
- local similarity 636
- LOD cloud 501
- LOD (Linking Open Data) 500
- logically equivalent 472
- logical reasoning 420
- logic in ancient Greece 429
- logic in ancient India 432
- logic machine 424
- logic program 482
- logic programming 475
- long term memory 171
- look ahead 245
- lookahead strategies 315
- lower bounding estimate 124
- LSI 699

M

- machine learning (ML) 416, 808
- machinery 420
- machine translation 702
- macro moves 221
- macro operators 75
- macroplanning 705
- makeNodes 37
- Manhattan distance 55, 653
- many sorted trees 832
- map colouring 305
- MAPGEN 397
- marginal probability 768
- marker passing 555

- Markov Decision Process (MDP) 779, 843
- Markov networks 775
- Markov process 779
- Mars Exploration Rovers (MERs) 397
- match 175
- match algorithm 179
- matching diagram 291
- Match-Resolve-Execute 175, 179
- materialism 105
- material value 244
- matrix diagonalization theorem 699
- Maven 272
- MAX 238
- max-domain-size 320
- maximum a posteriori (MAP) hypothesis 769
- maximum likelihood estimation (MLE) 810
- maximum likelihood hypothesis 769, 770
- max-span heuristic 383
- max sum algorithm 822
- MEA (means ends analysis) 175, 219
- meaning 451
- meaningful 420, 426
- Means Ends Analysis (MEA) 75, 219
- MEA strategy 221
- measurements 746
- mechanical reason 426
- memoize 342
- memory 413, 626
- memory based 625
- memory based reasoning 409
- Memory Organization Packets (MOPs) 566
- memory structure 566
- meronymy 687
- metadata 495
- meta rules 443
- Method of Elenchus 429
- metric spaces 653
- MEXAR 393
- microplanning 705
- MIN 238
- min conflicts 320
- mind body dualism 105, 426
- MinFill 312
- minimal candidate 742
- minimal conflict set 740
- minimal hitting sets 741
- minimal interpretations 714

- Minimax algorithm 247
- minimax rule 240
- minimax value 240
- minimum width orderings 311
- MinInducedWidth 312
- Minkowski norm 654
- mixed initiative 633
- modal logic 429
- modal operator 717
- mode assignment 739
- mode identification and reconfiguration 396
- model based 625
- model based diagnosis 736, 760
- models 416, 451, 721
- model theoretic semantics 428
- modified modus ponens 456
- modus ponens 439
- monotone property 137
- monotonic 731
- monotonic functions 734
- monotonic relations 727
- morphemes 683, 686
- morphology 683, 686
- most general unifier (MGU) 456
- moveGen 29
- Ms. Malaprop 561
- Multi-heuristic Best First Search 373
- mutex 344
- mutex relations 332, 335, 354
- MYCIN 185, 772

N

- Naive Bayes Classifier 811
- named entity recognition 702
- namespace 495
- Nash equilibrium 236
- National TSP 93
- natural deduction 440
- natural language 416, 491
- Natural Language Generation (NLG) 680, 705
- Natural Language Processing (NLP) 680
- Natural Language Understanding (NLU) 533, 680
- nearest insertion heuristic 94
- nearest neighbours 653
- negated goal 462
- negation as failure 481

- negative clause 472
- negative introspection 718
- negative path 581
- neighbourhood functions 96
- neighbourhood search 68
- NETL 560
- neural control 111
- neural network 105, 108
- Neurogammon 278
- neurons 106
- n-gram 638
- NOAH (networks of action hierarchies) 224
- node consistency 290
- node consistent 290
- no-function-in-structure principle 729
- nogood 761
- nogood inference rule 761
- nogood learning 341
- nominal attributes 836
- nonlinear planning 209
- nonmonotonic reasoning 709
- nonserializable subgoals 67, 221
- No-op action 334
- normalization 589
- N queens 36, 288
- number 506
- numeric attributes 637, 653
- Nyāyasūtra 432
- Nyāya-Vaiśeṣika 432

O

- objective function 68
- objects 607
- observations 739
- observation sequence 816
- observed behaviour 761
- Occam's razor 747
- one ply look ahead 245
- ontologies 545
- ontology 412, 490
- OPEN 32, 131
- open precondition 212
- operators 194
- OPS5 171, 173
- OPS83 185
- optimal policy 780

- optimal solution 119, 133, 164
- optimal strategy 241
- optimizing problem 391
- order crossover 98
- ordered symbols 640
- ordering links 210
- orders of magnitude reasoning 725
- ordinal representation 99
- overall condition 377, 380, 384
- overfit 832
- OWL 601

P

- paramodulation 470
- Pareto optimal 236
- parsing 689
- partial belief 800
- partial knowledge 732
- partial lookahead 319
- partially mapped crossover 97
- partial order planner 215
- partial order planning 209
- partial plan 212, 361
- partial solution 124
- partial syntactic analysis 702
- part of relation 551, 559
- part of speech 682
- part of speech tagging 683, 689
- path consistent 299
- path representation 91, 97
- pattern directed inference systems 169
- patterns 171
- PC-1 300
- PC-2 301
- PDDL 193, 196
- PDDL2.1 376
- PDDL3.0 388
- Perceptron 847
- Perceptron convergence theorem 850
- Perceptron training rule 848
- perturbation 89
- perturbation methods 95
- perturbation search 68
- phenotype 88, 110
- pheromone 112
- pheromone evaporation 114

- pheromone trail 113
- physical negation 747
- physical symbol system hypothesis 420
- plan applier mechanism (PAM) 570
- plan extraction 353
- planning 192
- planning as constraint satisfaction 352
- planning as satisfiability 358
- planning domain description language 193, 331
- planning graph 331, 333
- PlanningGraph 338
- planning in the real world 392
- planning problem 36, 202
- plans 411, 570, 779
- plan space planning 209
- plan trajectory 389
- plausibility 799
- poisoned rook 265
- policies 779
- policy 779, 783
- policy graph 783, 784
- Policy Iteration 788
- polysemy 682
- Porter's algorithm 687
- POS 682
- positional value 244
- positive clause 472
- positive introspection 718
- positive path 581
- possible worlds 349, 429
- POS taggers 689
- precision 694
- predicate completion 518
- predicate logic 448
- predicates 448
- predicate symbols 448, 451
- prediction 772
- preferences 192, 388, 391
- premises 421
- Priar 677
- principle of parsimony 740, 747
- prisoner's dilemma 235
- probabilistic context free grammars (PCFGs) 691
- probability 763
- probability density function 765
- probability distribution 764
- probability mass function 764

- Problem decomposition 156, 221
- problem descriptions 193
- problem solver 27, 171
- problem solving 26, 156, 407, 625
- problem solving agent 408, 544
- procedural knowledge 416
- procedural memory 414
- Prodigy 677
- productions 156
- production systems 168
- programming language 472
- Prolog 472, 478
- promotion 213
- proof 422
- proof by contradiction 446
- proof theoretic semantics 429
- PropagateProperties 597
- propagating probabilistic inferences 773
- propagation 733
- proper policy 781, 783
- properties 607
- propositional logic 425, 435
- protein sequence classification 813
- provable 422, 438
- pruning 145
- pure logic programming 478

Q

- QSIM 730
- qualitative behaviour 723
- qualitative description 722
- qualitative functional relations 731
- qualitative models 418, 721
- qualitative reasoning 418, 721, 736
- qualitative relations 375
- qualitative simulation 730
- qualitative state 721
- qualitative value 730
- qualitative variables 731
- quality of solution 41, 58
- quantity space 723, 724, 727, 732

R

- R1 169
- randomized search 115

- random walk 82
- rasa theory 526
- rational agent 410
- rational choice 235
- rationalistic approach 684
- rationality 235
- RDF document 496
- RDF Schema 498
- RDF triples 501
- reasoning 425, 513
- recall 694
- Recency 175
- recombination 112
- recombination and selection 101
- recommender systems 632, 633, 669
- reconstructPath 37
- recursive best first search (RBFS) 141
- refinement operator 120
- refinement planning 228
- refinement search 124
- refractoriness 175
- regression planner 365
- Reification 492
- reified 617
- reified elements 491, 546
- reified entity 506
- reinforcement learning (RL) 278, 842, 845
- relative diversity 670
- relaxation 561, 862
- relaxed plan 363, 369
- relaxed planning graph 367, 387
- relaxed planning problem 363
- relaxed temporal planning graphs 387
- relay node 144
- relevant action 200
- Remote Agent (RA) 394
- removeSeen 37
- repository 628
- representation and reasoning 547
- requests 570
- required concurrency 376
- resolution method 444, 445
- resolution proof 446
- resolution refutation 444, 462
- resolution rule 444
- resolve 175
- resolvent 444, 473

- resource 495
- resource description framework (RDF) 495
- Rete algorithm 180
- Rete network 179
- retrieval 631
- retrograde analysis 513
- Revise 292
- Revise-3 300
- reward 843
- reward function 779
- rhetorical structure theory (RST) 705
- river crossing puzzles 30
- robust plan execution 395
- role chains 597
- roles 585
- rollout 280
- rovers domain 232
- Rubik's cube 75
- rule based MT 703
- rule based systems 168, 185
- rule of inference 422
- rule of substitution 441
- rules 171, 173, 416
- rules of inference 438

S

- SAINT 158
- Samuel's Checkers program 234
- Sapa 380
- Sapir-Whorf hypothesis 540
- SAT 68, 358
- satisfaction 596
- satisfiability 436
- satisfiable formulas 436
- SATPLAN 358
- SAT problem 92
- savings heuristic 94
- scenario pattern matching 702
- scene labelling 296
- schema 549, 550, 558
- schemata 628
- Scrabble 270
- script applier mechanism (SAM) 565
- scripts 558, 563, 564
- search tree 30
- second order logic 483

- selection operator 89
- SelectValue 313
- self organization 860
- self organizing maps 861
- self referential sentences 483
- semantic knowledge 412
- semantic memory 567
- semantic net 559
- Semantic Network Processing System (SNePS) 614
- semantic parser 535
- semantic retrieval 618
- semantics 544
- semantic web 412, 494, 501
- Semcor 688
- semiotic interaction 104
- semiosphere 103
- semiotic interaction 112
- sense-deliberate-act 397
- senses 682
- sentence 450
- separation 213
- sequence alignment 143
- sequential retrieval 652
- serializable 208
- set of support 447
- shallow knowledge 625
- sharing experience 113
- SHOP2 227
- shortest path heuristic 582
- short term memory 171
- sigmoid function 83, 854
- similar 630
- similarity 54, 630, 642, 650, 663, 688
- similarity based retrieval 651
- similarity search 651
- simple conceptual graphs 618
- Simple Search 1
- 32
- simple temporal network (STN) 384
- simulated annealing 82, 862
- single point crossover 89
- singular value decomposition (SVD) 699
- skeptical reasoner 717
- skeptical reasoning 584
- skolem constant 458
- Skolem function 458
- skolemization 455

- SLD resolution 472, 473
- slot-filler 551
- Smart Executive 394
- snap actions 386
- SOAR 185
- Socratic argument 420, 431
- Socratic method 429
- soft constraints 391
- solution relation 303
- solution space search 68
- SOLVED nodes 163
- solving MDPs 787
- soma 106
- sound 439
- soundness 424
- sound rule of inference 439
- space complexity 41, 58
- SPARQL 501
- Sparse Memory Graph Search (SMGS) 145
- sparse solution path 146
- specialization 552
- specific boundary 827
- specificity 175
- spelling checker 685
- spy-swap problem 405
- squashing function 854
- SSP MDP 780
- SSS* 259
- SSS* algorithm 253
- stable expansion 718
- STAN 343
- START state 27
- state change verbs 530
- state consistency check 386
- state space 27
- state variable representation 355
- static evaluation function 54
- statistical machine learning 684
- statistical parsing 691
- steepest gradient ascent 61
- stemming 695
- stereotypical patterns 564
- stochastic actions 778
- stochastic hill climbing 83
- stochastic local search 92
- stochastic shortest path MDPs 780
- stopword removal 695

- story generation 576
- story understanding 565
- strategy 241
- string attributes 637
- STRIPS 193, 203
- strongly i-consistent 302
- structural adaptation 673
- structural CBR 632
- structured representations 545
- structure matching 589, 591
- structures 550
- structure to function 728
- subgoal interaction 204
- subgoal ordering 208
- subgoals 208
- subjective probabilities 763
- substitution 455
- subtour selection crossover 100
- successors 29
- summary generation 704
- supervised learning 809, 832
- supervised training 847
- surface realization 706
- Sussman's anomaly 208
- SVD 699
- swarms 103
- syllogism 422
- syllogisms 431
- symbol attributes 640
- symbolic integration 161
- symbolic representation 420
- symbol manipulation 425
- synonymy 700
- synonymy 687
- synthetic reasoning 676
- systematic 230

T

- Tabu search 72
- tabu tenure 72
- TALE-SPIN 576
- Tamagotchi 412
- target function 823
- tautological equivalence 441
- tautological implication 439
- tautology 436

- taxonomic symbols 641
- taxonomies 545
- taxonomy 552, 554, 642
- TBox 588
- TD-Gammon 278, 280, 845
- teachable language comprehender 560
- temporal constraint networks 329
- temporal constraints 384
- temporal difference (TD) learning 278, 843
- temporal Graphplan (TGP) 378
- temporal planning 374
- temporal planning graph 382
- TemporalRPG 387
- term frequency 695
- terminological reasoning 588
- term weighting 696
- Tesauro 843
- text classification 811
- text summarization 704
- textual CBR 632
- tf-idf score 695
- thematic abstraction unit (TAU) 577
- thematic organization packets (TOPs) 676
- theme 572
- theorem prover 428
- theorem proving 443
- Thing 588
- threat 212
- threshold based retrieval 666
- threshold function 108
- threshold similarity 631
- Tic-Tac-Toe 242
- time complexity 40, 59
- timed initial literals (TIL) 385
- timeline 398
- tokenization 695
- top-down parser 690
- topic fusion 704
- topic identification 704
- Towers of Hanoi 220
- training 847
- training examples 810
- trajectory constraints 388, 391
- translation model 704
- travelling salesman problem (TSP) 91
- T-REX (Teleo-Reactive Executive) 397
- triangle inequality 664

- trigrams 639, 685
- trihedral objects 296
- triples 495
- truth maintenance 752
- truth maintenance system (TMS) 745, 752
- TSP 95, 124
- TSP-ACO algorithm 115
- TSPLIB 92
- TWEAK 217

U

- UAV 397
- unary constraint 290
- uncertainty 708
- unconditionally independent 776
- Undercut 283
- underestimating function 164
- unification algorithm 455, 457
- unifier 455, 477
- unifies 455
- uniform resource identifier (URI) 495
- uninformed 50
- unique name assumption (UNA) 713
- unique name axioms 517
- unit preference 447
- universal grammar 540
- universal instantiation 452
- universal quantifier 448, 451
- universal truth 433
- universe of discourse 450
- unordered symbols 640
- unsatisfiability 436
- unsatisfiable formulas 436
- unsupervised learning 809, 847
- utility 629, 650, 779
- utility accumulating function 779
- utility based semantics 509
- utility distinguishability 630

V

- vague predicates 509
- valid argument 421
- valid formulas 436
- validity 436
- valid plan 203

- valuation function 783
- valuations 437
- value function 779
- value iteration 789, 791
- valuespace 495
- variable neighbourhood descent 69
- variable ordering 320
- vector space model 694
- version space 827
- Viterbi algorithm 822
- VLSI TSP 93
- vocabulary 629, 632

W

- Wayland 676
- weak methods 31
- Web 3.0 501
- web of data 501
- web ontology language 601
- well formed formula 424
- width 306
- winning strategy 241
- Wordnet 561, 682, 687
- WordNet similarity 697
- word phrases 538
- word sense disambiguation (WSD) 561, 682, 688
- word senses 539
- working memory 171
- working memory elements (WMEs) 171
- World TSP 93
- Wumpus world 232

X

- XCON 169, 185

Z

- zero sum game 239
- zugzwang 264